UiO : Department of Physics
University of Oslo

An Attitude Determination and Control System for CubeSTAR

Master Thesis

Cecilie Bjelbøle

February 1, 2013

Abstract

This thesis describes the development of the Attitude Determination and Control System (ADCS) for CubeSTAR. CubeSTAR is a small 2-U unit CubeSat satellite developed by students at the University of Oslo. In this thesis a module card for the ADCS has been developed. The complete hardware has been developed for the fulfilling of the entire ADCS. The hardware consists of magnetometer, MEMS gyroscope, coil drivers and an FPGA for the necessary computational power for the complete ADCS. The gyroscope and magnetometer have been calibrated. The coil drivers have been tested and verified.

Acknowledgments

This thesis is the fulfilling of the Master of Science in Electronics and Computer Technology at the Department of Physics, University of Oslo. This work was carried out from January 2012 to January 2013 under the supervision of Associate professor Torfinn Lindem and PhD candidate Tore André Bekkeng.

I'm very grateful to Torfinn Lindem for given me the opportunity to work with this interesting topic. I would like to thank Lindem for his support and guidance during this work and for his work on the CubeSTAR project. Tore Andé Bekkeng, thank you for everything. You have guided and motivated me through this whole project. I couldn't have asked for more.

A special thanks to Halvor Strøm and the rest of the guys at the Electronics lab for all the help in the PCB production. I also want to thank Espen Trondsen for sharing all his practical wisdom.

To all of my fellow students, this would not been the same without all of you guys. To Bent, David, Elling and Andres at room 303, thanks for all help, discussions and nonsense during this time. To my parents, thank you for all you love and support through everything. A special thanks to my boyfriend Harald, for sticking up with all my ups and downs. You are amazing.

Contents

Nomenclature

ACS	Attitude Control System
ADC	Analog to Digital Converter
ADCS	Attitude Determination and Control System
ADS	Attitude Determination System
ARM	Anistotropic Magnetoresistive
CoCom	Coordinating Committee for Multilateral Export Controls
COTS	Commercial off-the-shelf
CRAM	Configuration Random-Access Memory
ECEF	Earth Centered, Earth Fixed
ECI	Earth Centered Inertial
EMF	Electromotive Force
ESR	Equivalent Series Resistance
GPS	Global Positioning System
I2C	Inter IC bus
IGRF	International Geomagnetic Reference Field
LE	Logic Element
LEO	Low Earth Orbit
LQR	Linear-Quadric Regulator
LUT	Look Up Table

m-NLP	multi-Needle Langmuir Probe
MEMS	Micro-Electro-Mechanical Systems
MUX	Multiplexer
NAROM	Norwegian Centre for Space-related Education
NSC	Norwegian Space Center
OBC	On-Board Computer
PFM	Pulse-Frequency Modulation
POD	Picosat Orbital Deployer
PWM	Pulse-Width Modulation
RISC	Reduced Instruction Set Computer
TWI	Two Wire Interface

Chapter 1

Introduction

This thesis describes the design and development of the Attitude Determination and Control System (ADCS) on the CubeSTAR satellite. The satellite's attitude is it's orientation in space. This attitude needs to be determined and controlled, hence the name Attitude Determination and Control System. CubeSTAR is a student satellite developed at the University of Oslo, following the CubeSat standard.

1.1 Background and Motivation

In December 2008 the CubeSTAR project was initiated by the Department of Physics at University of Oslo, together with the Norwegian Centre for Space-related Education, (NAROM) and the Norwegian Space Centre (NSC). The STAR project is a collaboration between space physics and the electrical engineering group at the University of Oslo. STAR stands for Space Technology And Research.

CubeSTAR will carry a scientific experiment as its payload, a multi-Needle Langmuir Probe (m-NLP) developed at the University of Oslo. The m-NLP is measuring electron density in ionospheric plasma. Measuring the electron density is of interest for space weather monitoring over the polar caps. Communication and positioning satellites will benefit from space weather reports. These satellites suffer from amplitude and phase distortion when the solar wind activity is high. The instrument is capable of measurements at a much higher resolution than todays standards with a spatial resolution of 1 meter. The Langmuir Probes operates without the need to know the spacecraft potential and the electron temperature [4]. The optimal orbit altitude for the payload is somewhere between 400-650 km. This altitude is called LEO orbit (Low Earth Orbit). CubeSTAR is planned to launch in 2014.

The other main goal of the CubeSTAR project is to give interesting student projects for master thesises. Therefore the satellite is built from scratch by students

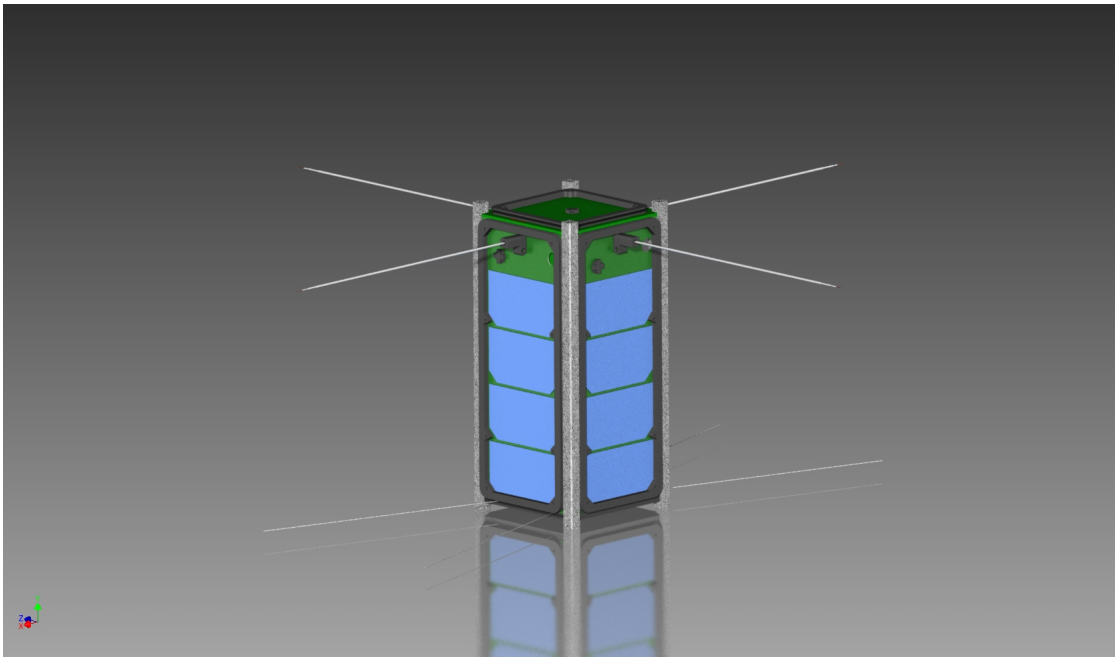


Figure 1.1: A 3D model of the CubeSTAR satellite with antennas, langmuir probes, sun sensor and magnetorquers.

and employees at the University of Oslo. The satellite is divided into the following subsystems:

- Communication [29], [10]
- Electrical Power System (EPS), [20]
- On-Board Data Handling (OBDH)
- Attitude Determination and Control System (ADCS), [21], [27]
- Payload, [5]

A ground station for communication to the CubeSTAR satellite have been developed in addition to the subsystem of the satellite. See [31] and [30] for more information.

1.2 The CubeSat standard

The CubeSat standard was developed by California Polytechnic State University (Cal Poly) and Stanford University's Space Systems Development Lab. They developed the CubeSat standard to help universities with a standard and give access

Figure 1.2: Engineering model of the CubeSTAR satellite. Photo: Marius E. Hauge

to space by providing launch opportunities through CubeSats programs. The standard provides both electrical and mechanical specifications for the satellites. A P-POD (Poly-Picosat Orbital Deployer) was developed for deployment of the CubeSat. The CubeSats are being launched as an additional payload on rockets carrying larger satellites. This kind of standardization makes the launch of a CubeSats cheaper than other regular launches of satellites with no standardizations. The original CubeSat had a size of $10cm \times 10cm \times 10cm$, called a 1-U satellite. CubeSTAR has a size of $10cm \times 10cm \times 20cm$, this is equivalent to a 2-U in the CubeSat standard. More information about the CubeSat standard and the CubeSat community can be found at [22].

1.3 Attitude Determination and Control

When the satellite is deployed into space, the Langmuir Probes and antennas need to be pointed in the correct direction. The attitude determination and control system is a crucial part of the satellite. For the payload to be successful, the Langmuir Probes need to be pointing in the front of the spacecraft in the velocity direction. The probes need to be in the front because of the turbulence created by the satellite itself when it moves through the ionospheric plasma. A system for determining the attitude and controlling the satellite, have to be in place for the scientific mission to be a success. The attitude determination and control system can be divided into two groups; the attitude determination part (ADS), and the

attitude control part (ACS). An attitude accuracy of ± 10 degree is set as the goal for the entire attitude determination and control system.

1.4 Previous Work on ADCS

This thesis is part of a bigger ongoing project; the development of the satellite CubeSTAR. The main goal for this thesis is to contribute to the projects advancement by further developing of the ADCS subsystem on CubeSTAR. This thesis is a continuation of the work of Stray [27] and Rensel [21]. Stray [27] delivered a thesis about attitude control and it was decided to use an FPGA for the ADCS. This work contributed with the initial design parameters for the magnetic coils. Rensel [21] contributed to the project with the first version of the ADCS card. Two different gyroscopes and a magnetometer were tested and verified. This was implemented with using a microcontroller. A coil winder machine were made for production of magnetic coils.

1.5 Goals for this Thesis

The main goal for this thesis is to provide sufficient hardware for the determination and control system on the CubeSTAR satellite. This will include a new version of the ADCS module card. The card will include attitude sensors and control actuators, together with a sufficient computational power for the complete ADCS. Sensors and magnetic coil drivers should be tested and verified. The ADCS card will be developed as a slave for the On-Board Computer in the satellite and be able to communicate with this subsystem. The developed module card will be developed with the purpose of implementing ADCS and a sun sensor at a later stage in the project.

Chapter 2

Attitude Determination and Control System

Satellites carry antennas and/or payloads that need to be oriented in a specific direction. Therefore, it is desired to have control over the satellite's attitude; the satellites orientation in space. The goal of the attitude determination is to know the satellite's orientation in its body frame with respect to a reference frame. These expressions of different frames are described below. For our mission the m-LMP needs to be in front of the spacecraft relative to the velocity direction. For communication, is it important to know and to control the directions of the antennas. This chapter will give an introduction to the basics behind an attitude determination and control system. Several methods for representing the attitude are presented along with common sensors and actuators for ADCS. A short overview is given to the ADCS modes, b-dot, determination and control mode.

2.1 Attitude representation

2.1.1 Reference frames

A reference frame is usually a Cartesian coordinate system where the frame moves with the center of mass of an object. The object can be the spacecraft or a planet. The attitude between two reference frames can be described by a rotation matrix between the two frames. These different frames can be used to assess how different objects are oriented to each other. The choice of an axis system is closely related to the satellite's task [24]. A reference frame is a three dimensional Cartesian coordinate system, denoted by $\mathcal{F}_b$. It consists of a triad of orthogonal unit vectors, denoted by the same letter $\hat{\mathbf{b}} = \{\hat{b_1}\hat{b_2}\hat{b_3}\}^T$.

Satellite Body Frame

A body frame or satellite body frame is defined by the spacecraft's three unit vectors centered in the spacecraft center of mass. The CubeSTARs satellite body frame is denoted by $\mathcal{F}_b$. The z_b axis is defined as the mechanical top of the satellite. x_b is pointing towards the back plane and y_b completes the right hand rule.

Earth Centered Inertial (ECI) Frame

Earth centred inertial frame, as it states, is fixed at the centre of the Earth denoted $\mathcal{F}_i$. This is a non-rotating frame, where z_i points towards the Earths geographical north pole and x_i points at the vernal equinox point. Vernal equinox is the point where the plane of the Earth's orbit around the sun crosses the Equator at March equinox.

Earth Centered, Earth Fixed (ECEF) Frame

Earth Centered, Earth Fixed (ECEF) Frame, $\mathcal{F}_e$ where z_e is directed towards the north pole. ECEF frame is a rotating frame fixed to the Earth. x_e points to the Earths 0° latitude and 0° longitude and y_e completes the right hand rule.

Satellite Orbit Frame

A satellite orbit frame, $\mathcal{F}_o$ has it's origin in the center of mass of the satellite. The x_o axis points in the velocity direction of the orbit.z_o points towards the Earths center, nadir and the y_o completes the right hand rule. This frame is the reference frame for the satellite body frame and the attitude can be described as the transformation that maps the vectors from the reference frame to the body frame, [32].

2.1.2 Rotation Matrix

The attitude of a spacecraft is represented by defining the rotation from the body vectors u, v, w to the vectors in the reference frame. The vectors from the reference frame are given by numbering: 1, 2, 3. This 3×3 matrix is called the direction cosine matrix, rotation matrix or the attitude matrix. The direction cosine name comes from the reason that the matrix consist of cosines of the angles between the body vectors and the reference-frame vectors. The rotation matrix from the reference frame to the body frame is defined:

$$A = \begin{bmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix} \tag{2.1}$$

The u vectors are the unit vector components from u along the three axes 1, 2, 3 in the reference frame. This applies to the v and w vectors as well. This matrix will map a vector from the reference frame to the body frame. Let's say we have the vector $\mathbf{a} = [a_1 a_2 a_3]^T$. $\mathbf{a}$ is expressed in the body frame by the following operation:

$$[A]\mathbf{a} = \begin{bmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \mathbf{u} \cdot \mathbf{a} \\ \mathbf{v} \cdot \mathbf{a} \\ \mathbf{w} \cdot \mathbf{a} \end{bmatrix} = \begin{bmatrix} a_u \\ a_v \\ a_w \end{bmatrix} = \mathbf{a}_B \tag{2.2}$$

Where the vector $\mathbf{a}_B$ is the original vector $\mathbf{a}$ mapped into the body frame from the reference frame.

2.2 Sensors

To know the spacecraft's attitude several sensors is used. The most common sensors are mentioned here and the sensors used on CubeSTAR are described further.

2.2.1 Gyroscope

A gyroscopic sensor, or a gyro is a device measuring rotation based on the principle of the conservation of angular momentum [8]. Classical gyros consists of a mass free to rotate in all directions which can be used to either keep, or measure the orientation. The rotating mass is mounted with a spin axis free to rotate in one direction. Two outer gimbals enable the possibilities of free rotation. The rotating mass is considered the output axis of the gyro and is mounted perpendicular to the spinning axis. When an angular velocity is present an output tourque proportional to the input angular velocity will be observed, this is called a precession of the gyro.

MEMS technology makes it possible to fabricate smaller gyroscopes compared to the classical ones, without the need for large gimbals and a spinning axis. Vibrating gyroscopes are the most frequently used microgyroscopes today, [12]. Simplified the MEMS gyro has a suspended, vibrating mass in a frame, instead of a mass free to rotate. MEMS technology makes it possible to utilize a gyroscope on a small CubeSat. This technology has been driven by the consumer market to produce smaller and cheaper electronics. Many MEMS gyroscopes are now found on the market integrated in small IC's with digital output and at a low price. Vibrating gyroscopes work on the principle of Coriolis effect. A mass, m in a rotating system will feel a force perpendicular to the direction of the velocity it is subjected to. Newton's law, $F_c = ma_c$, where the Coriolis acceleration is given in

this case $a_c = 2V \times \Omega$, lead to the expression of the Coriolis force, F_c

$$F_c = -2mV \times \Omega \tag{2.3}$$

Where m is the mass, Ω the angular rate of the system and v the velocity of the moving mass. The Coriolis force acting on the mass will be in the direction perpendicular to the velocity, given by the right hand rule. This tells us the displacement of the mass is dependent of the angular rate subjected to the system. [12] Gyroscopic sensors will give the output in angular rate, and thus integration must be done in order to obtain an attitude. A drawback with the integration is that a bias on the gyro will add up. Because of this, gyroscopes are not well suited for obtaining aboslute attitude, but is used for adding accuracy to the rest of the attitude sensors in the ADCS.

InvenSense ITG 3200

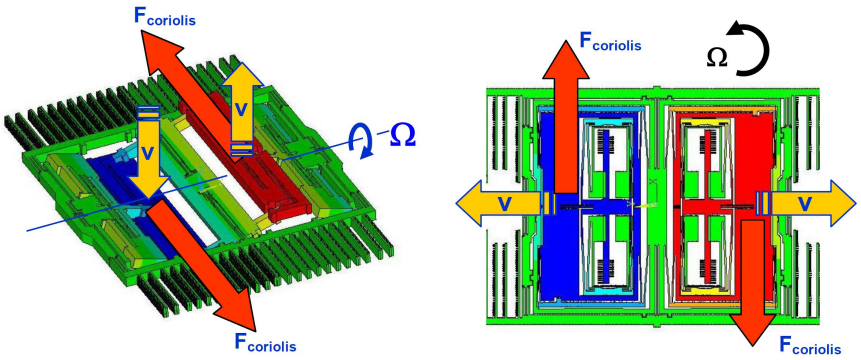
The InvenSense gyro is the sensor chosen for this project from the previous work done. The gyro was compared to a higher end gyroscope, the SAR100 from Sensonor and after calibration both the sensors gave good results. More information is found in [21]. Because of cost and availability the ITG3200 sensor was chosen. The InvenSense gyro is produced for the consumer market with all three axes on one chip. The ITG3200 gyroscope has a digital output communicating over I2C and a full scale range of ±2000 deg/sec. The resolution of the gyroscope is 0,07 $rate/LSB$ where $rate = deg/sec$. This is considered as a sufficient resolution for the gyroscope onboard CubeSTAR.

To manage all three axis on one chip the x and y-axis is placed perpendicular to each other and the z-axis has a bit working principle comapared to the x- and y-axis. For all three axes, there are to suspend masses mounted together in such a way that they move in opposite direction to each other. [13]

2.2.2 Magnetometer

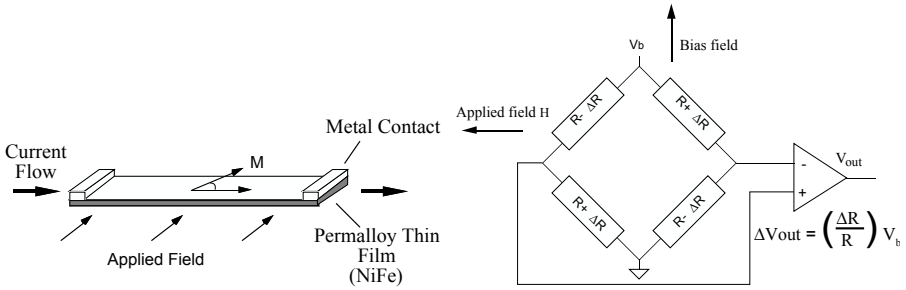
Magnetometers are widely used as spacecraft attitude sensors. [32] They are known to be reliable, lightweight, have low power consumption and can operate over a large range of temperature. A magnetometer measures the magnetic field vector. The Earth's magnetic field strength decreases with distance, a $1/r^3$ dependence. This makes the magnetometer most suitable for satellites in the LEO orbit. The precision of the magnetometer is given by the uncertainties in the knowledge of the Earth's magnetic field. The Earth's magnetic field is strong and well defined in LEO, which makes a magnetometer a common sensor on CubeSats.

A magnetic field measurement from the magnetometer needs to be compared to the Earth's magnetic field. A model of the Earth magnetic field with the name



(a) The X axis. *(b) The Z axis*

Figure 2.1: Working principle of the ITG 3200 gyroscope. The X and Y axis are based on the same principle. [23].



(a) Magnetoresistive *(b) Wheatstone bridge*

Figure 2.2: Magnetoresistive sensor properties [18]

International Geomagnetic Reference Field (IGRF) is common to implement. This is a standardized model with a precision of one tenth of an nT. Today's model is a 13th order formula. The IGRF is generally revised every five years.

Honeywell HMC5883L

In this thesis the HMC5883L magnetometer from Honeywell is being used. It is a 3-axis magnetometer with a small size and digital interface. The HMC5883L is based on Anisotropic Magnetoresistive (ARM) technology. AMR exist in ferrous materials such as NiFe. When a magnetic field is applied perpendicular to the current flow in the material, there will be a change in resistance.

The sensor is built up of magnetoresistive strips connected to each other forming a Wheatstone bridge as shown in Figure 2.2b. This four resistors are placed in the same direction, but they are connect for the current to flow in different direc-

tions. This will cause the resistance to increase in two resistors, while it decreases in the other two. The change in magnetic field is linear to the applied voltage, V_b:

$$\Delta V_{out} = \frac{\Delta R}{R} V_b \tag{2.4}$$

$$\Delta V = SHV_b$$

where S is typically

$$S = 3\frac{mV}{V/Oe}$$

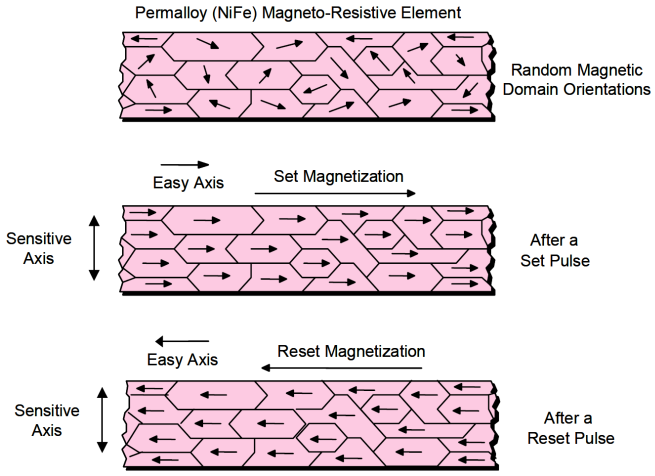


Figure 2.3: The magnetic orientation in permalloy thin film [1]

The sensors thin NiFe strips consist of a small domain with magnetic orientation. This orientation could be permanently changed to arbitrary directions by a large magnetic field, while a smaller magnetic field will only temporarily change the magnetic domains. The accuarcy of the sensors is highly dependent upon a uniform direction of the magnetic domains in the thin film. When in use, the external magnetic field is normally only changing the magnetic domains temporarily. In operation the current is flowing in the same direction, but the external magnetic field is changing in strength and direction affecting the resistance and hence, the output. For correct operation it is important that the magnetic domains are directed perpendicular to the current flow. This is done prior to measuring by a *set/reset* circuit. This circuit will apply a strong magnetic field over the Permallow thin film. A *reset* is for use when an inverted magnetic field is applied. *Reset* sets the magnetization orientation in the opposite direction as *set* wich leads to an inverted output response from the magnetometer. This *set/reset* circuit is implemented in the HMC5883L magnetometer.

Honeywell has launched a new magnetometer called HMC5983 wich could easily replace todays existing magnetometer. This magnetometer has a higher sampling rate, 220 Hz against 160 Hz and a temperature compensated output. The only adjustment requiered in the existing design is to tie pin 4 to VDD for I2C communication.

2.2.3 Sun Sensor

A sun sensor is a commonly used sensor for attitude determination in satellites [3] [32]. The sun can be treated as a point source of light at infinite distance, with parallel rays of light. The sun can easily be distinguished among other light sources in space. Sun sensors can be made in many different ways, from analog to more complex digital sensors. Analog sun sensors measure current variations in a photocell. The output from the analog sun sensor is proportional with the cosine of the angle of incidence of the solar radiation. Digital sun sensors have a geometric shape that is illuminated. Often a geometric shape is made to form a shadow on the imaging device. New generation digital sensors utilize an imaging device with a mask placed in front of it.

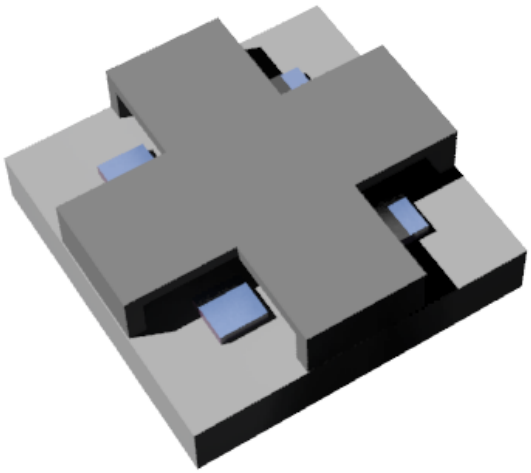


Figure 2.4: Design of the sun senor that will be implemented on CubeSTAR at a later stage.

The sun sensor on CubeSTAR is a sensor designed and manufactured at the University of Oslo. The sensor will consist of four photodiodes placed together to form a square. To know the angle to the sun a structure is placed around the diodes for a controlled shadow. This sensor is a two axis sun sensor. The sensor will be placed on all sides of the satellite, except on the bottom where the antenna is located. By placing a sun sensor on each sides of the satellite, the entire field of view of the satellite is covered. The four diodes will be placed together with an ADC and communication to the ADCS will go through an I2C bus. The signal processing for the sensor will be located at the FPGA on the ADCS card. The implementation of this sun sensor will happen at a later stage in the project.

2.2.4 Horizon Sensor

A horizon sensor, as the name implies, detects the horizon of the Earth. A horizon sensor can be seen as a form of an earth sensor. To a satellite orbiting close to Earth the Earth is the brightest object besides the sun and covers almost half the sky. [32] It is nor sufficient enough to detect only the presence of the Earth for crude attitude determination. Hence there is often a need for detecting the horizon of the Earth. The horizon sensor is often implemented by an IR detection which senses the warmth of the Earth compared to the colder space. The sensor will detect changes in intensity when it crosses the horizon [32].

2.2.5 Star Sensor

Star sensors locate stars in the frame of the spacecraft and compare this to known star directions. They are complex and often quite large and expensive systems, and thus not often seen on CubeSats. The advantage of the star tracker is its high accuracy as an attitude sensor [32]. The sensors consist of sensitive cameras and a catalogue of known stars. For three-axis determination there is sufficient for measuring stars with their known location in interial space. The star sensor needs to detect different stars from each other. This is done by considering their magnitude, light spectra and position relative to each other. Today star sensors are often based on CMOS image sensors and with the use of effective OBC makes it possible to use this kind of sensor on a CubeSat. One problem with the star tracker is its sensitivity to higher angular velocities which causes smearing of the star image. According to [3] new modern star sensors can handle angular rate up to a limit of 10 deg/min.

2.2.6 GPS

A Global Positioning System (GPS) can be used for attitude determination of satellites. One way of determining the attitude with GPS is by measuring the phase variations in the GPS carrying signal at two antennas. Utilizing GPS in LEO orbit is possible, but a big challenge may be restrictions set by Coordinating Committee for Multilateral Export Controls (CoCom). The GPS CoCom prevents the GPS device of working when moving faster than 1852 km/h or an altitude of 18 km. This limit is set for prevent the use of intercontinental ballistic missiles. Several GPS systems have been implemented in CubeSats previously [3].

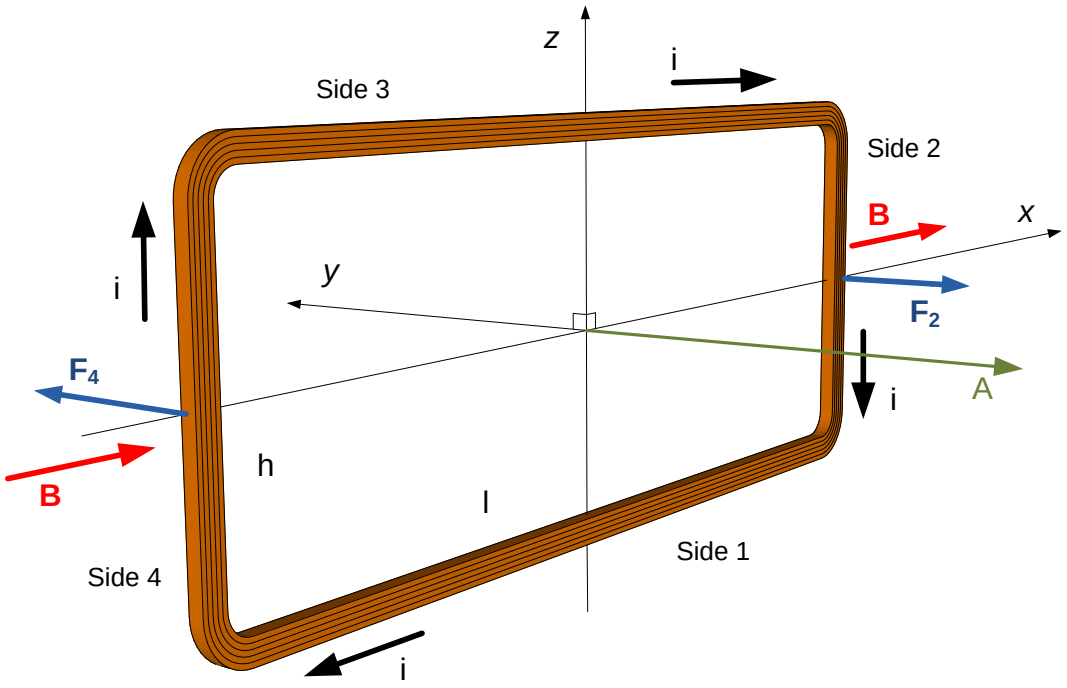


Figure 2.5: The magnetic force F, in a magnetic coil with the current i [21].

2.3 Actuators

When the spacecraft's attitude is known it needs to be controlled. This can be achieved by the use of actuators or passive stabilization methods.

2.3.1 Magnetorquers

A magnetorquer is an electromagnetic coil creating a dipole magnetic moment. Magnetorquers are active stabilization controls. The coils can be made with or without a metallic core. Because of weight considerations the magnetic coils for CubeSTAR will not have a magnetic coil. The magnetic dipole is created perpendicular to the face area of the coil and will therefore try to align with the earth's magnetic field, see Figure 2.5. To control the spacecraft three magnetic coils are placed on each side of the spacecraft perpendicular to each other. The torque created by the coils is described by a cross product:

$$T_coils = m_coils \times b_b \tag{2.5}$$

where b_b, is the geomagnetic field. This makes the spacecraft controllable in only two dimensions. The geomagnetic field is not homogeneous, and can be

considered periodic. Therefore it can be controllable in three dimensions with magnetorquers as only actuator.

A magnetic force F, acting on a cunducting wire in a magnetic field, B can be described :

$$F = is \times B \tag{2.6}$$

Where i, is the current and s, is the length and direction of the wire. Figure 2.5 shows a $l * h$ coil with a current i flowing through it. The rectangular loop is lying in the xz-plane centered over origin, a magnetic field B is applied to it. This is uniform and parallel to the x -axis. Because side 2 and 4 are perpendicular to the magnetic field they will have magnetic forces present, $F2$ and $F4$. Side 1 and 3 are in the same direction to the magnetic field and no forces will work on them. The magnitude of $F2$ and $F4$ is given by:

$$F_2 = F_4 = ihB \tag{2.7}$$

$F2$ and $F4$ are equal in size because the magnetic field and current are the same for the both of them. These forces are creating a torque which is trying to rotate the coil around the z-axis. The magnitude of the torque generated by forces working perpendicular is given by: $T = |rF|$, where T is the torque and r is the distance between the forces. r in our loop is l and $F = F2 = F4$. This give:

$$T = lihB = AiB \tag{2.8}$$

Where A is the area of the loop. By adding more turns in our loop, the forces are multiplied by a number of turns:

$$T = niA \times B \tag{2.9}$$

n, i and A are defined as the magnetic dipole momentum, $\mu \equiv niA$. With a steady state current and an air core the magnetic dipole moment is given:

$$\mu = nIA \tag{2.10}$$

Ohm's law is known: I = U/R. The voltages in the coils are not a variable since we do not want a voltage regulator for the magnetorquer circuit. The resistance in the coils is affected by number of turns:

$$R = (nl\sigma(T))/a_w \tag{2.11}$$

where l is the circumference, a_w is cross section area of the wire and μ is the material resistivity of the conductor, which is dependent of the temperature, T. given by: $(T) =_0 [1 + (T - T_0)]$ By combining μ and R in ohm's law, we get:

$$\mu = nA\frac{U}{nl\sigma(T)/a_w}$$

$$\mu = A\frac{Ua_w}{l\sigma(T)} \tag{2.12}$$

From 2.12 we see that the magnetic moment μ is not affected by number of turns in the coils. But indirectly it does, because the number of turns affect the face area A of the coil. The number of turns affect the resistance and hence the power consumption. The coils are designed with consideration to get a magnetic dipole moment between $60mAm^2$ and $100mAm^2$. They are designed to get the biggest available face area. This will be considered as constant. The only parameter left to adjust is the wire dimension a_w and the number of turns since the voltage is fixed. Available space is constant and the temperature is not controllable.

2.3.2 Momentum Wheels

A momentum wheel is a momentum exchange device, such as magnetourquers and control moment gyros that will not be discussed further in this thesis. A Momentum wheel is a rotating mass which stores angular momentum. They consist of massive wheels trading momentum back and forth between the satellite and the momentum wheels for control of the satellite. The momentum wheels utilize Newton's third law of rotating devices. To rotate the spacecraft in one direction the momentum wheel is set to spin in the opposite direction. For 3-axis control three momentum wheels are placed perpendicular to each other. They are often used when the need for high attitude control accuracy is present. They are very heavy and is quite expensive for CubeSat projects. [32] [24]

2.3.3 Permanent Magnet

A permanent magnet is a passive method for stabilizing the spacecraft. This is an easy and effective method that is commonly used. The permanent magnet on board will try to align itself, and also the spacecraft, to the earth's magnetic field. This will provide for two axis of stabilization. Magnetic hysteresis rods are used with this permanent magnet for damping oscillation around the spin axis. A magnetic hysteresis rod is made out of a soft magnetic material. The clear disadvantages with this method is that when the satellite is crossing the poles the magnet will change direction and cause the satellite to tumble.

2.3.4 Thrusters

Thrusters are devices that shoots out mass in one direction and by this creating a force in the opposite direction and they are reffered to as mass-expulsion control systems. Thrusters requires fast response and a great accuracy of your system. This will often result in a more complex hardware system. Because thrusters

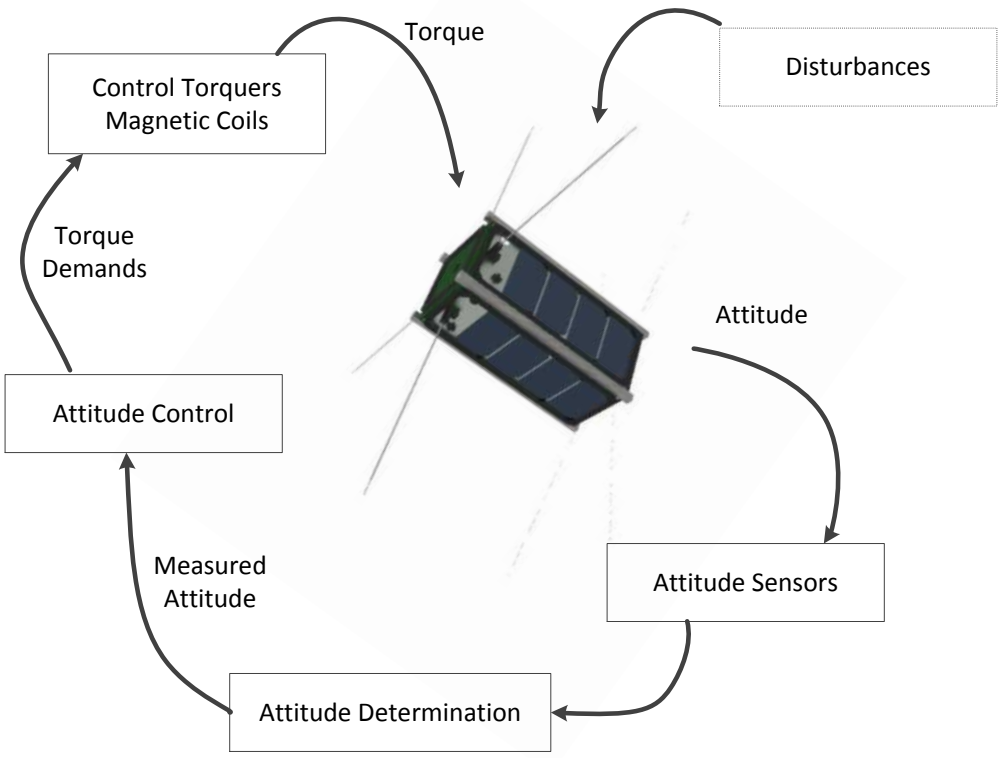


Figure 2.6: Illustration of CubeSTAR in space. The satellites attitude changes because of distrubances and torquers applied to it. The attitude determination is calculated with the help of several sensors. This information is given to the attitude control which rotates the satellite to a wanted attitude.

uses fuel they have a limited life span and are also quite heavy. Thrusters are a commonly used actuator method for bigger satellites but they are too heavy and expensive for use on small cubesats. [32]

2.4 ADCS on CubeSTAR

The ADCS can be perceived as two different systems, the Attitude Determination System (ADS) and the Attitude Control System (ACS). The data from the sensor is used in the ADS to compute the satellites present attitude. This information is given to the ACS that rotates the satellite to the wanted attitude. This process is shown in Figure 2.6. The control torques and disturbances affect the satellites

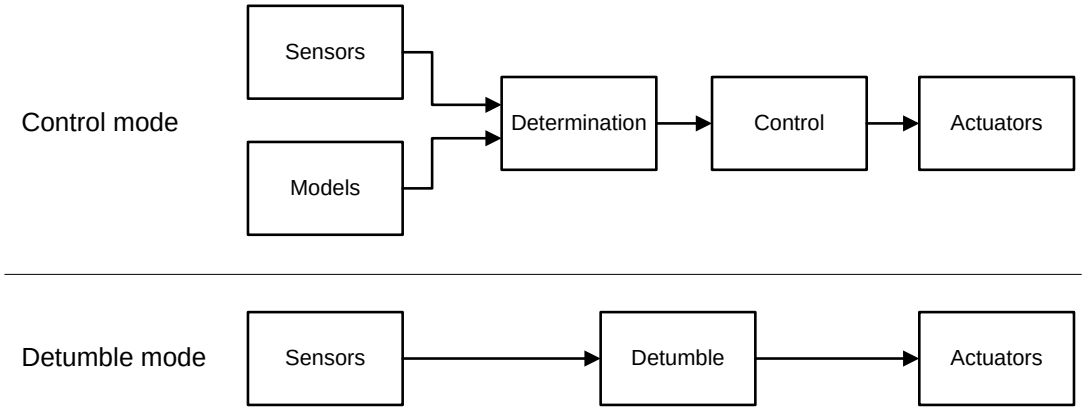


Figure 2.7: The control mode and detumble mode. The main goal for the detumble mode is to lower the angular rate of the tumbling satellite. The control mode is the most complex.

attitude and the cycle starts all over again. The control torques on CubeSTAR are three magnetic coils mounted perpendicular to each other. The three coils x, y and z are mounted on their respective axis seen in the satellites body frame.

The ADCS on Cubestar is divided into two modes, detumble mode and control mode. The control mode is shown in more detail in Figure 2.6. The detumble mode will only be active once, when the satellite has been deployed from the launch vehicle. The accuracy limit in a ADCS system is determined by the combination of processing procedures and the hardware on board the spacecraft, [.]

2.4.1 B-dot

When the satellite has been deployed from the POD the satellite will possess a relatively high angular rate, we say the satellite is tumbling. The satellite can also start to tumble because of disturbances and deployment of mechanical parts, for instance antennas. Before starting the attitude determination algorithm it is necessary to reduce the angular rate. This is requierd for several reasons. The advanced controller may not be stable in high angular rates and the different sensors may not be able to give correct measurements to the attitude determination algorithm and the determined attitude may be off [27]. Because of this a simpler controller will be implemented as a detumbler for the satellite.

A b-dot controller is implemented in CubeSTAR as a detumbler controller. The b-dot controller is very simple and is not dependent upon complex system to work. It tries to minimize the derivative of the magnetic field vector. The b-dot controller sets up a magnetic field on the magnetorquers that's proportional to the derivative of the magnetic field. This causes the satellite to act like a compass

needle following the geomagnetic field. The b-dot controller is given:

$$\mathbf{m} = -K\dot{\mathbf{b}}_b \tag{2.13}$$

where $\mathbf{m}$ is the magnetic output moment, K is a positive constant gain, and $\dot{\mathbf{b}}_b$ is the derivative of the measured magnetic field in the body frame. $\dot{\mathbf{b}}_b$ is given:

$$\dot{\mathbf{b}}_b = \mathbf{b}_b \times \omega_b^{ib} + \dot{\mathbf{b}}_i \tag{2.14}$$

Eq. 2.14 simplifies to

$$\dot{\mathbf{b}}_b \approx \mathbf{b}_b \times \omega_b^{ib} \tag{2.15}$$

From simulations in [27] the b-dot managed to reduce the angular rate from 0.1 rad/s to 0.002 rad/s in three orbits with a gain, K of 10000.

The b-dot algorithm has been implemented in the ADCS system by [27] and [21] and it is a fall back system for the Attitude Determination and Control System. For many satellites it is crucial for the payload that the attitude control system works perfectly. For CubeSTAR measurements can still be taken as long as the satellite knows when the langmuir probes are in turbulent areas. If the attitude is known, but not fully controlled the data from turbulent areas can be discarded.

2.4.2 Control Algorithm

The main goal of the ACS is to orient the spacecraft to a desired attitude when the current attitude is known. The magnetic coils control the satellite and the coils are controlled by a control algorithm. A small satellite can be controlled in a broad range of methods. From [27] it was recommended for CubeSTAR to use the constant gain LQR controller for the attitude control. The LQR problem is finding the input that minimize a given criteria. In [27] a PD controller, LQR constant gain controller and the LQR periodic gain controller was investigated. The constant gain LQR method requires less memory and is easier to implement than the periodic LQR control and the PD controller.

2.4.3 Attitude Determination

The ADS goal is to determine the attitude for the spacecraft. This is done by using several sensors to obtain accurate measurements. The sensor data is compared to a wanted reference frame. Several different types of algorithms can be used for the attitude determination process. One of the most common filters is the Kalman filter. The filter has a fading memory characteristic and a possibility to adjust the filters confidence in its estimate. This kind of filter is very useful in attitude determination where constant tracking of a changing attitude is required.[32]

Kalman Filter

Kalman introduced a recursive filter in 1960 and has been one of the most widely used estimation algorithms since. The Kalman filter is a filter which estimates the state of a dynamic system from measurements observed over time with noise and other inaccuracies. The filter produces statistically optimal output that tends to be more accurate than the use of a single measurement alone. If the system and measurement is linear and a statistical description to the system and measurements, the Kalman filter is the optimal linear minimum mean-square-error estimator for Gaussian systems. Only the estimated state from the previous output and the current measurement are needed to calculate the next output. For a good introduction to Kalman filters for engineers see [3]. For a reprint of the original article see [16]. [32] also has a good chapter on this subject.

The Kalman filter predicts a state based on a linear system described by the process equation:

$$\mathbf{x}_{k+1} = \Phi_k \mathbf{x}_k + \Lambda \mathbf{u}_k + \Gamma \mathbf{w}_k \tag{2.16}$$

where $\mathbf{x}$, is the state vector. The state vector is a sum of the previous state, the control input and noise. Φ_k relates the state at the previous time step and k to the state at the current state. Φ_k is called the state transition matrix. $\mathbf{u}$ is the control input, where Λ is the control input matrix. $\mathbf{w}$ is the process noise assumed zero-mean Gaussian noise, and Γ is the process noise matrix. The measurement equation is described:

$$\mathbf{z}_k = H_k \mathbf{x}_k + \mathbf{v}_k \tag{2.17}$$

where H is the observation matrix and $\mathbf{v}$ is the measurement noise, assumed to be zero-mean Gaussian white noise. To implement the Kalman filter it needs to be discrete. The Kalman filter time updates are given:

$$\overline{\mathbf{x}}_{k+1} = \Phi_k \widehat{\mathbf{x}}_k + \Lambda \mathbf{u}_k \tag{2.18}$$

$$\overline{P}_{k+1} = \Phi_k \widehat{P}_k \Phi_k^T + \Gamma_k Q_k \Gamma_k^T \tag{2.19}$$

The $\overline{\mathbf{x}}$ is the apriori estimate, the estimate before the measurements is taken into consideration. The $\widehat{\mathbf{x}}$ is the posteori estimate, where the measurement are taken into consideration. P is the uncertainty of the filters estimate.

The measurement equations are responsible for updating the measurement into the apriori estimates given in 2.18 and 2.19. The measurement equations are given:

$$K_k = \overline{P}_k H_k^T (H_k \overline{P}_k H_k^T + R_k)^{-1} \tag{2.20}$$

$$\widehat{\mathbf{x}}_k = \overline{\mathbf{x}}_k + K_k(\mathbf{z}_k - H_k \overline{\mathbf{x}}_k) \tag{2.21}$$

$$\widehat{P}_k = (I - K_k H_k)\overline{P}_k \tag{2.22}$$

where the Kalman gain is given by K, dependent on the priori covariance, the measurement noise and the measurement noise covariance matrix, R. The next state, posteori state estimate $\widehat{\mathbf{x}}$ is generated by taking the measurement $\mathbf{z}$ into consideration. At the end a posteori error covariance matrix is estimated with the apriori estimate of P, $\overline{P}$. The process is then repeated from Equation 2.18-2.22 to update the next time and measurement pair to new estimates.

Chapter 3

Electronic Design

This chapter describes the design and development of the ADCS card. The first version of the ADCS card was made by K.Rensel. Several sensors are utilized on both versions and thus there are many similarities in parts of the electronic design.

3.1 System Overview

The ADCS card consists of an FPGA with communications lines to all of the sensors; a magnetometer circuitry, gyroscope circuitry, magnetourquer driving circuits and interface to a sun sensor.

All internal communication is using I2C lines. All sensors have their own I2C line for redundancy. The sun sensor consist of several sensors placed on five sides of the satellite, all of these communicate together on one I2C bus. This bus, with pull up, is implemented on the ADCS card. To the FPGA from the OBC there are two I2C buses, A and B. There are also two lines, RX and TX, for UART communication to a computer. To communicate over UART a RS232 driver are needed. This is located on an external card, the mini back plane card developed by [21].

3.2 PCB realization

Two PCBs that are going to be the hardware platform for the Attitude Determination and Control System were realized, from now on referred to as the ADCS card. A previous version was made by Rensel [21]. He made a choice and recommendations of sensors in his thesis and other solutions for the electronic design that I have used further. The main purpose for making a new version of the ADCS-Card is to implement a FPGA to handle the attitude determination and control part. As mentioned, two PCBs were produced in this project. There are only minor

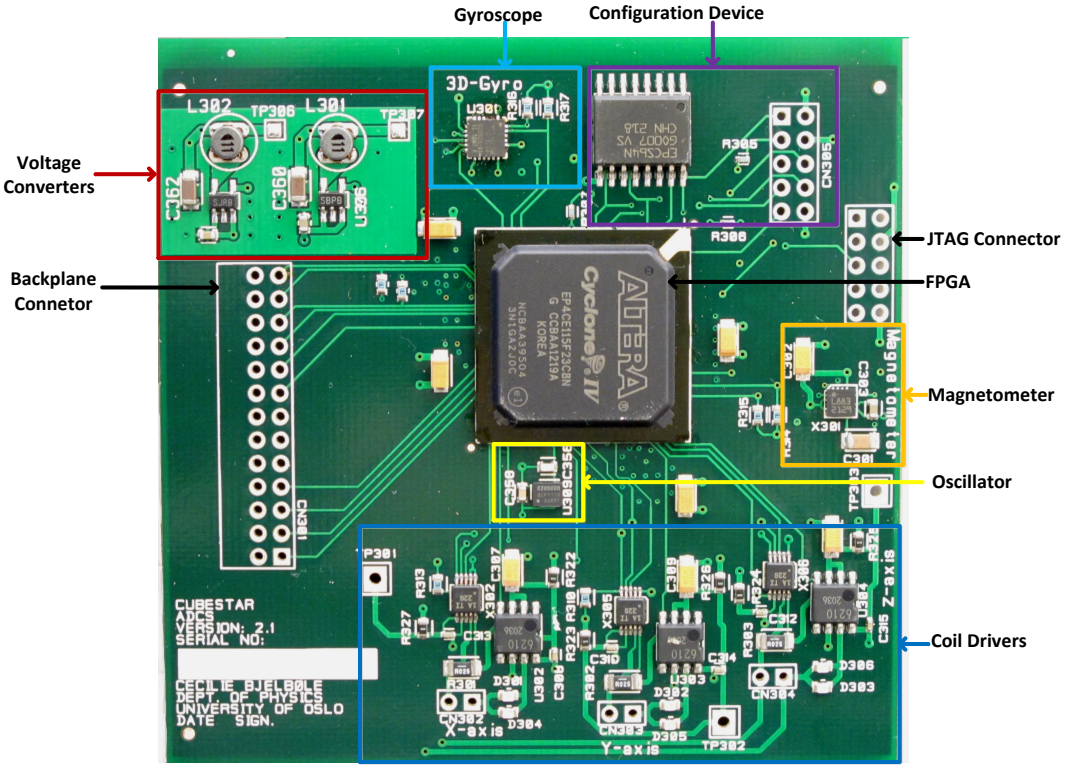


Figure 3.1: The latest version of the ADCS card with description of the functional blocks.

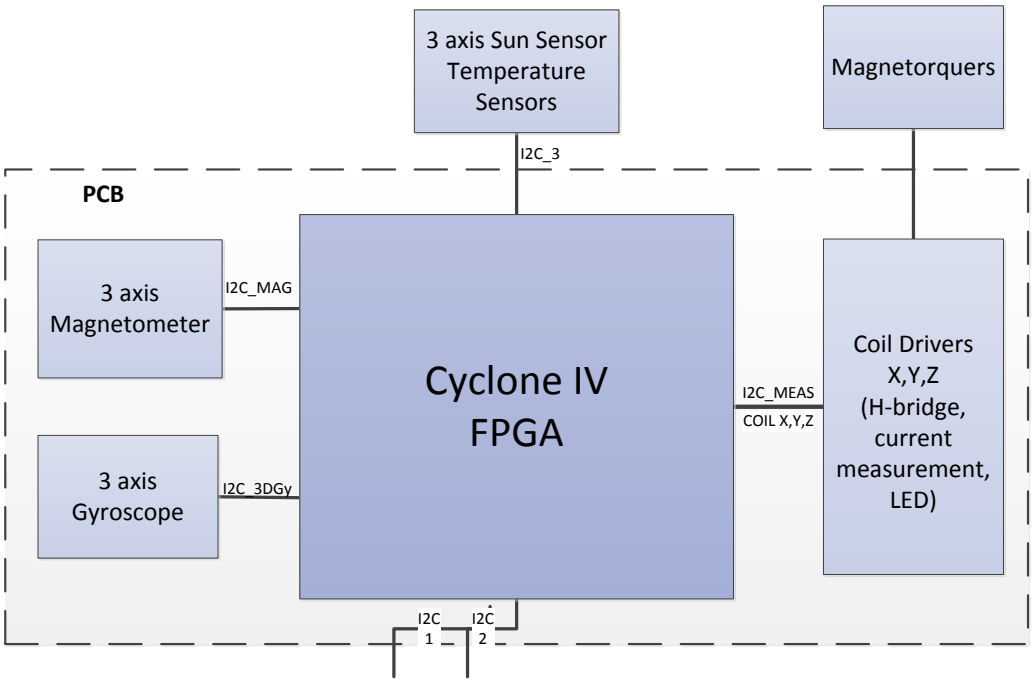


Figure 3.2: System overview of the ADCS card with sensors and actuators. The Cyclove IV FPGA is the master of all the sensors and actuators on the ADCS card (marked as PCB), together with the sun sensor and magnetorquer placed on the side panels of the satellite. The sun sensor and magnetorquer are mounted on side panels on the outside of the satellite. The sun sensors are placed next to a corresponding temperature sensor, all of these sensors communicate over the same I2C bus.

differences between these two. I will only discuss briefly the difference between them.

The first version used the Cyclone IV FPGA with a core voltage of 1,0 V. This FPGA was replaced with the 1,2 core voltage version. One of the main reasons for doing this was the ability to use the implemented configurational error detection with the user mode error detection. Errors may happen in configurational random-access memory (CRAM) due to ionizing particles. The built-in implementation of this reduces the need for external circuitry. This user mode error detection is not implemented in devices with a 1,0 V core voltage.

In the first version it was intended to use two LDO regulators to convert the input voltage to 2,5 V and 1,0 V. These two circuits were never mounted because of wrong footprints. An external voltage supply was used instead. The use of voltage regulators that are power efficient is the best solution for both heat and the power budget. The new version of the ADCS card utilize two switching buck converts for the voltage conversion from the battery voltage to 1,2 V and 2,5 V.

3.2.1 Physical Dimension

The physical dimensions of the PCB is given by the physical constrains of the CubeSat standard [22]. A module template with physical dimensions and some general construction rules was made by the electronic workshop. A four layers PCB card was used for the ADCS card. Routing of the FPGA can be a difficult task, largely depending on the number of I/O to the chip. The I/O was made to a minimum, this made the routing easier and there were no use for a 6-layer PCB. Placing the I/O pins in the outer circle of the FPGA made the signal routing better. The PCB consist of signal layers as shown in Figure 3.3; top signal layer, ground layer, power layer and bottom signal layer. Only through hole vias are used. The PCB was designed using Zuken CADSTAR software.

3.2.2 Ground Plane

Most PCBs designed today consist of one or more ground layers. A ground plane is a common way to provide a stable reference throughout the card for all components [9]. Ground is the common reference level wich all components refer to as 0 V. This makes it important to have the same reference potential throughout the PCB. The path taken for return current is dependent on the frequency. At high frequencies the return current takes the path with least inductance. For lower frequencies the return current takes the path of least impedance. Ground loops may cause serious error in a system by changing the electrical potential. If a current drawn by a device goes in a large loop before returning to the device, it may cause noise or be susceptible to noise from the surroundings. It is therefore important to have a

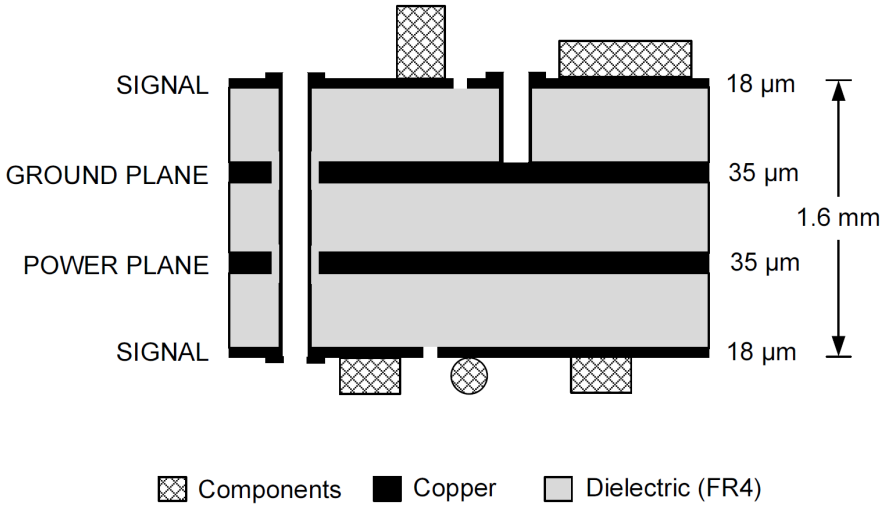


Figure 3.3: 4 layers PCB with plated through via and signal layers.

low impedance path for the return current. A ground plane will reduce the loop area of the potential ground loops and hence reduce the problem. [8],[9]

Digital circuits contribute to noise because of their switching behaviour. Analog circuits are on the other hand, the most vulnerable to noise. Splitting the ground plane into analog and digital ground planes is common [9]. In this design only digital components are used and there is no need for splitting the plane in digital and analog ground. On board the satellite there is only one point of reference to ground and splitting it into analog and digital will at the end will probably not make any difference. A section of the top layers is also used as a pseudo ground layer for minimizing noise from the voltage converters.

3.2.3 Power Plane

On the ADCS card there are three available voltages. A 3,3 V power bus is going into the module card. The power plane is split into two different voltages, 1,2 V and 2,5 V. The 1,2 V level is placed underneath the FPGA which makes the 1,2 V easily accessible to the core of the FPGA. Two larger capacitors are placed in both ends of the area for stable bypassing of the power layer. The rest of the voltage layer is dedicated to the 2,5 V reference that supply the rest of the components on the PCB.

3.2.4 Decoupling and Ground Loops

To reduce the noise and hence the ground loop areas, decoupling capacitors are used. Digital components draw current unevenly in pulses. When an IC draw power in form of switching, there will be a small current going through the ground and power plane. A current spike will therefore cause a voltage spike that is corresponding. This will cause transient voltage drops in the supply voltage and that transmit to the ground plane and seen as noise. This will lead to voltage drops if the power source is not able to supply current fast enough. Decoupling capacitors are used to make a low impedance path from the IC's power supply and ground [8]. When a digital circuit switches there will be a current flowing from the IC directly to ground and also a current flowing from the IC through the load to ground. The bypass capacitors function is to deliver current immediately to the IC. Values on bypass capacitors are determined by the specific design. The transient response from these components must often cover a large range of frequency and load range. Therefore several different values are often used together to secure the best performance. Larger values are often placed near group of components to cover a lower range frequency. Smaller bypass capacitor is placed between the power and ground pin of the IC for the higher frequencies. For higher frequencies ceramic capacitors are often used and for lower frequencies tantal capacitors are often used. For best performance the bypass capacitor is positioned as close to the IC as possible. When it's not possible to place the capacitors close to both the ground and power pin, the ground pin is prioritized. Several practical examples can be seen in [9].

3.3 Choice of Hardware Platform

Choice of Technology

An FPGA is chosen as the hardware platform for the ADCS on CubeSTAR. To the author's knowledge, COTS FPGAs have not been common on regular larger satellites and launched into space. Microcontrollers have been the most common hardware platform for CubeSats previously. Today we see a shift of this to a more use of FPGAs in CubeSats.

If a microcontroller is going to be used in a more high precision system, a more complex microcontroller will be chosen. With the use of an FPGA a more scalable system is achieved, only the parts needed are implemented. A custom system can easily be made with an FPGA compared to the more standardized solutions from the microcontroller manufacturer.

Ionizing radiation can cause unwanted effects such as bit flipping in state of memory cells in semiconductor devices. This is the main drawback with utilizing

an FPGA in space. Because a COTS FPGA is not that common on satellites, it is of interest to see how this technology will perform in space.

Choice of FPGA

From the previous work of [27], an FPGA with embedded multipliers is needed. At first the Stratix III from altera was considered, also recomended from the work of Stray. This is a high performance FPGA contamining a large number of embedded memory and multipliers; from 9x9 multipliers to 36x36 multipliers. The Stratix III FPGA also has high static power consumption. The smallest option for Stratix III is 23x23 mm.

The Cyclone family may be a better choice for CubeSTAR. This is the lowest cost and the FPGA family with the lowest power consumption from Altera. Both the Stratix and Cyclone family have a built in configuration error detection for check of single-event upset.

The Cyclone IV EP4CE115 with a F484 package is chosen as the hardware platform for the ADCS system. This is the largest FGPA in the Cyclone IV E series, with 532 9x9, 266 18x18 and 266 embedded multipliers. It is also possible to implement soft multipliers by using the M9K memory blocks as LUTs. This will increase the number of available multipliers. The package has a size of 23x23 mm and is the smallest size of this device. It was considered to go for a smaller package of 19x19 mm, but then either the FPGA size or the number of multipliers would be reduced considerably. The alternative device is the EP4CE75. The number of logic elements in all of these alternatives will be sufficient for the ADCS algorithms.

At this stage there is no knowledge about the determination algorithm CubeSTAR is going to use. Because of this, the choice of hardware platform has been taken by best guesses. It is therefore crucial that the attitude determination algorithm is developed with the hardware implementation in mind. The choice is taken with knowledge about the attitude control algorithm that is going to be used. By knowing that the control algorithm takes 360 of 532 9-bits multipliers, there are still a lot of multipliers left for the determination algorithm. Maybe the smaller FPGA EP4CE75 also will have a large enough amount of multipliers, but at this stage an overestimate is done. The package chosen of 23x23 mm is small enough to be placed on the ADCS PCB card and also has a large enough pitch (1 mm) that makes it feasible for in-house mounting at the electronics workshop.

3.4 Hardware Architecture

3.4.1 I2C

Communication to and from the ADCS-card is done over a I2C (Inter IC) bus. This bus is also implemented on the card for comunication to all the sensors included in the ADCS system. I2C was developed by Philips Semiconductors (NXP Semiconductors) for inter-IC communication. I2C is also reffered to as TWI. I2C consist of two biderectional wires, SDA and SCL. I2C is a master-slave system that can consist of mulitple masters and slaves. All devices, slaves are connected to the bus by a unique address. Both the slave and master can operate as a receiver and transmitter. The master controls the clock line and is the only one capable of initializing a data transfer. Both the master and slave can control the SDA line.

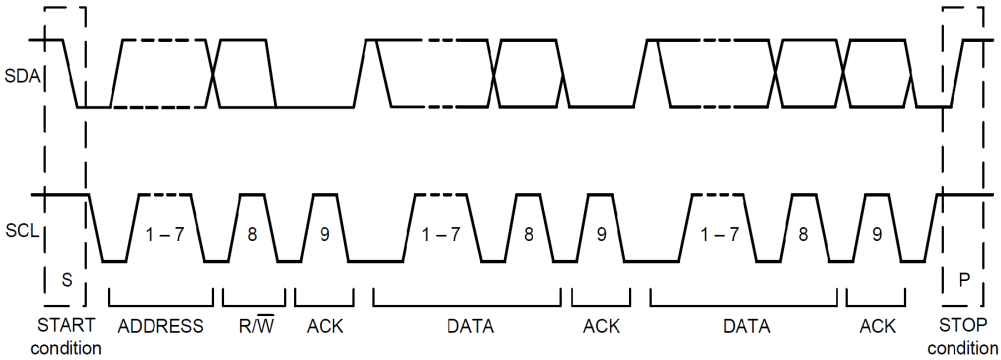


Figure 3.4: Complete data transfer on the I2C bus. [13]

To start a data transfer the master sends the start condition, pulling the SDA line low while the clock line (SCL) is high for a while. Then the master sends the slaveaddress (7 bit) and a read or write bit. The corresponding slave aknowledges the address and send an ACK (acknowledge) bit. The ACK bit is holding the SDA line low in the ninth clock cycle. Depending on the sensor, after the address is sent the next step is to send the register addres. Then the slave will know what register to read or write to, depending on the read/write bit in the first bit transfer. After each byte of data the slave or master needs to send an ACK bit. The master can terminate the transfer by transmitting a stop condition. The stop condition occurs when the master release the SDA line high while the SCL line is already at high level.

3.4.2 3-Axis Gyroscope Circuitry

A 3-axis gyroscope from InvenSense was chosen as the gyroscope for CubeSTAR. This gave quite good results after calibration compared to the SAR150 gyroscope investigated in [21]. ITG3200 is a small, cheap and a COTS component. It's developed for use in consumer applications for instance motion based gaming consoles, location based services, point of interest and toys. The chip has 3-axis on one chip and a digital output with an I2C interface. The chip has internal 16 bits analog to digital converters for each axis and a sensitivity of 14,375 LSB/(deg/sec). The entire configuration of the gyro is done with I2C communication. Figure 3.5 shows the internal block diagram of the gyro.

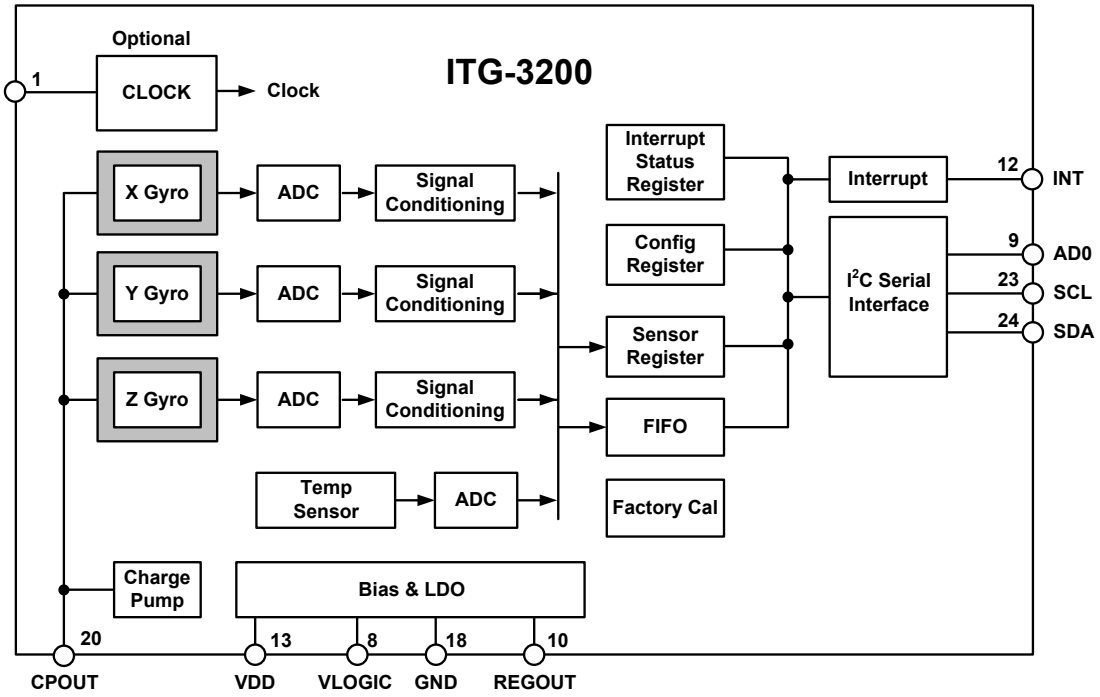


Figure 3.5: The gyro ITG-3200 internal block diagram [13]

3.4.3 Magnetometer Circuitry

The magnetometer on the ADCS card is a magnetometer from Honewell, HMC5883L. This is a 3-axis magnetometer with a theoretical accuracy of 1-2 deg. HMC5883L comes in a fine pitch 16 pins LCC package. The sensor needs only three additional capacitors to work properly. On the I2C bus where the FPGA is the master controller the magnetometer is acting as a slave. Figure 3.6 shows the principal block

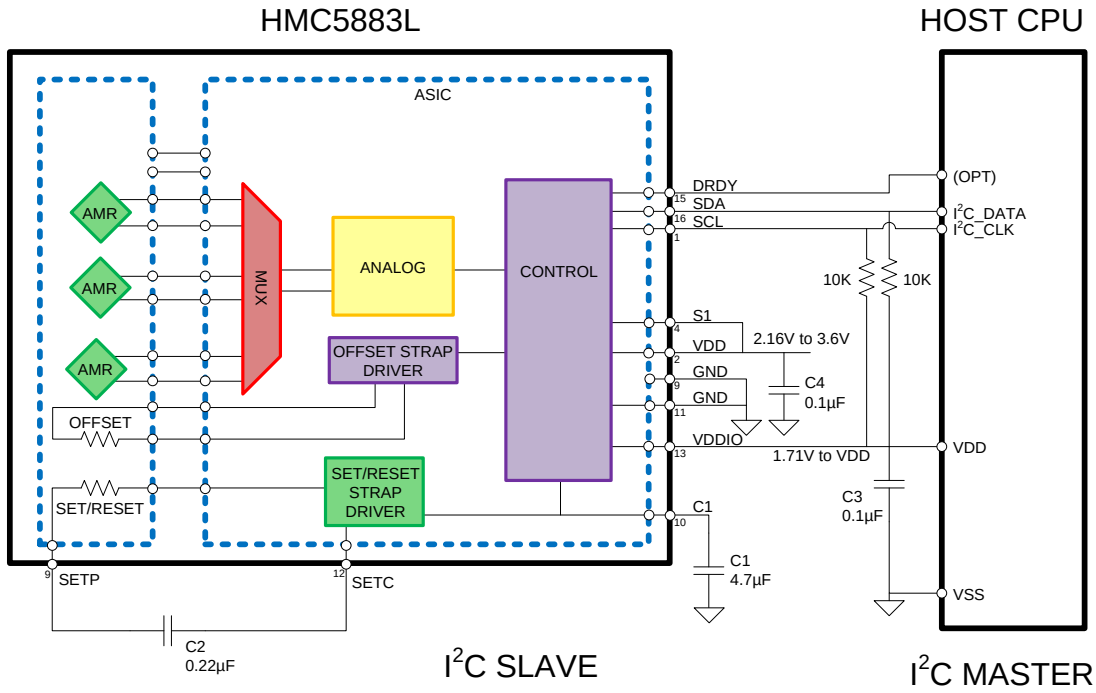


Figure 3.6: Internal block schematics of the magnetometer HMC5883L and external circuitry [28].

schematic of the sensors functions with it's external requirements. The sensor can operate over a supply voltage range from 2,16 V - 3,6 V. It's possible to use a lower voltage for the IO interface allowing communication to components using a lower voltage IO. In this circuit the IO lines operates at 2,5 V. Supplies for the sensor and IO interface are therefore connected together. A DRDY pin on the chip works as an interrupt pin notifying when data is ready to be read from the sensors data registers.

3.4.4 Magnetourquer Driver Circuit

To be able to control the direction and amount of current flowing through the magnetorquers, a magnetourquer driver circuit is implemented. This circuitry consists of three H-bridges with corresponding current monitors for each of the three coils. An H-bridge consists of four transistors, where two of them are connected to an output. One of the two outputs is connected to VCC and the other to GND. This enables the H-bridge to control the output direction by connecting the output transistors individually to either VCC or GND. The control of the current flowing through the magnetorquers is given by the Pulse-Width Modulation principle,

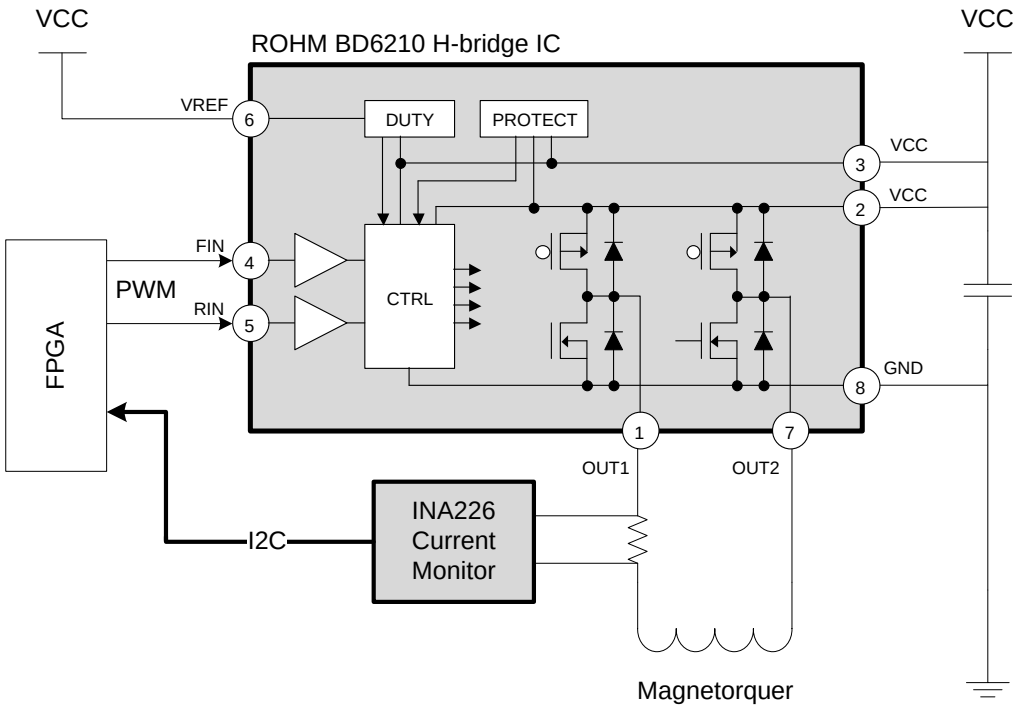


Figure 3.7: The H-bridge, BD6210, with external circuit to control the magnetorquers. Based on [21] and [11]

switching on and off. Controlling the on-time (the pulse width) of the signal, the current is regulated accordingly. The implemented H-bridge on the ADCS card is the BD6210F from ROHM Semiconductors. This IC was tested in [21]. A current sensing circuit is implemented to monitor the current flowing through the coil, this is an IC from Texas Instrument INA226. A schematic of the BD6210F with external circuitry is shown in Figure 3.7. The only change in this circuit from the first version of the ADCS card is the implementation of a new current sensing circuit communicating over I2C.

Controlling the Magnetourquer

The BD6210F consist of four MOSFET transistors connected together with diodes placed across each transistor. The BD6210F can deliver up to 0,45 A at the output of the IC. The magnetic coils are inductors. This requires some special considerations because of the inductors unwanted properties. When current is flowing through the magnetic coil, it creates a magnetic field in the coil. When the current flow is turned off, the magnetic field will try to resist the change in current and the magnetic field will induce a back EMF(Electromotive Force). If

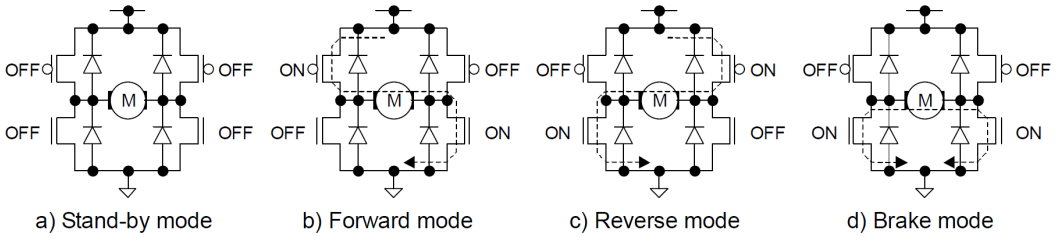


Figure 3.8: The four basic modes of the H-bridge [11].

the coil is disconnected and there's a magnetic field present in the coil, the induced current do not have anywhere to go and high voltages can be present. The diodes in BD6210F make sure the current has a way to pass the transistors, even if they are all turned off.

The control of the magnetorquers consists of determining the amount of current and the direction. The amount of current is controlled by the PWM principle as mentioned above. The direction of the current is controlled by changing the H-bridge's input signal. Table 3.1 gives an overview over the BD6210F's operating modes. The four different modes are shown in Figure 3.8.

In a forward mode (mode b) the current will flow through one transistor and through the magnetorquer (illustrated as a motor, M), and through the transistor on the opposite side before current flows to ground. The reverse mode sends the current in the opposite direction. The brake mode, d is used to brake a motor in rotation. The back EMF generated in the coil is now in the opposite direction and the motor is breaking.

For the control of the magnetorquers we want to utilize reverse and forward mode, but with the ability to use PWM. Mode e and f are used when running and d while in idle mode.

Current Sensing

A current sensing circuit is implemented on the ADCS card as shown in Figure 3.7. In the previous version of the ADCS card the current sensing circuit implemented was not functional as this IC was intended to be placed on the high-side of the load. The INA226 is a bi-directional current and power monitor for high- or low-side measurement, which makes it very flexible to use. With its ability to communicate over I2C it eliminates the need for an ADC between the current monitor and the FPGA. INA226 calculates the current by measuring the voltage drop over a shunt resistor. Current is calculated by multiplying the measured shunt voltage with a calibration value. The calibration value is the product of maximum expected current and the shunt resistor value. A 25 $m\Omega$ resistor were chosen. It was later

Table 3.1: Functional description of the H-bridge. Mode e, f and d are used.

	FIN	RIN	VREF	OUT1	OUT2	Operation
a	L	L	X	Hi-Z*	Hi-Z*	Stand-by (idling)
b	H	L	VCC	H	L	Forward (OUT1 > OUT2)
c	L	H	VCC	L	H	Reverse (OUT1 < OUT2)
d	H	H	X	L	L	Brake (stop)
e	PWM	L	VCC	H	$\overline{\text{PWM}}$	Forward (PWM control mode A)
f	L	PWM	VCC	$\overline{\text{PWM}}$	H	Reverse (PWM control mode A)
g	H	PWM	VCC	$\overline{\text{PWM}}$	L	Forward (PWM control mode B)
h	PWM	H	VCC	L	$\overline{\text{PWM}}$	Reverse (PWM control mode B)
i	H	L	Option	H	$\overline{\text{PWM}}$	Forward (VREF control)
j	L	H	Option	$\overline{\text{PWM}}$	H	Reverse (VREF control)

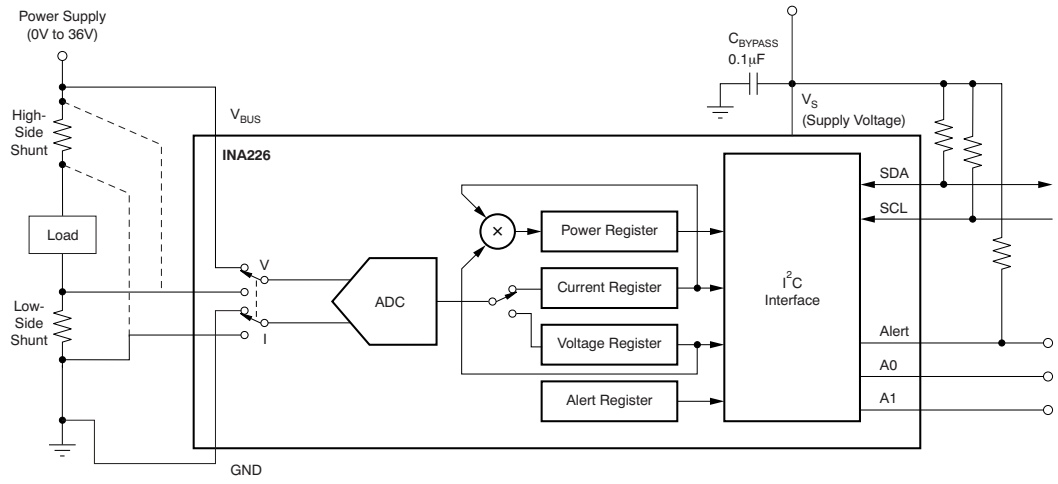


Figure 3.9: Internal block schematic over the current sensing circuit INA226 with external components. The INA226 is able to measure current on both high-side and low-side.

discovered that this will only give a shunt voltage value of maximum 2,5 mV (with a max current of 100 mA). The input range of the current sensor is $\pm$ 82 mV and thus a max range of only $\pm$2,5 mV will give a bad resolution. A new shunt resistor of 1,5 Ω was chosen instead and gave a more reliable result.

3.4.5 FPGA

FPGA or Field Programmable Gate Array, is a reprogrammable digital chip. FPGAs consist of programmable logic capable of implementing the same function as most other custom ASIC could perform. The smallest cell in a FPGA is a logic element (LE). LE's may be configured to perform complex combinational functions or simple functions like AND gates and OR gates. Basically a LE consist of a small LUT, a programmable register and a MUX [6]. To design the digital logic, the VHDL programming language was used. VHDL is a Hardware Description Language used to describe the structure and behaviour of digital electronic.

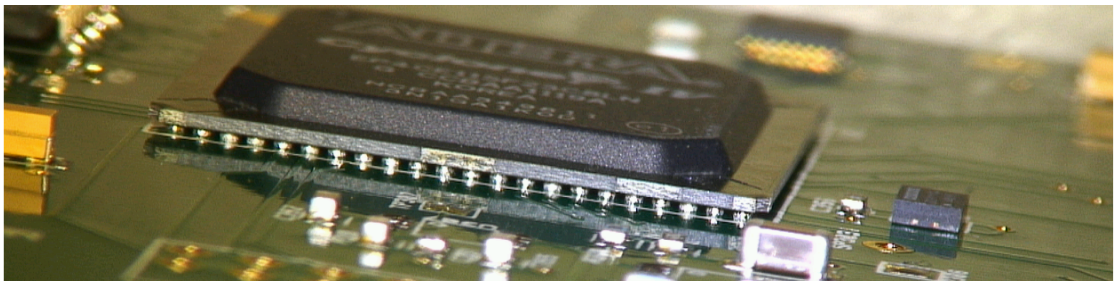


Figure 3.10: Cyclone IV FPGA mounted on the ADCS card.

Nios II Core

The presence of a processor core in an FPGA enables the possibility for hardware and software design to be built into one single device. The presence of a soft core makes the ADCS card very versatile. The Cyclone IV FPGA on the ADCS card includes a Nios II soft core. Nios II is a 32-bit embedded-processor with a RISC architecture. The core is a configurable soft IP core, in contrast to a microcontroller. With the Nios II core it is possible to specify and generate a custom core for the specific project. It is possible to set up the Nios II core in 3 different configurations; Nios II/f (fast), Nios II/s (standard) and Nios II/e (economy). Nios II/s is implemented on the ADCS card. Altera's first version of the Nios core was named Nios. In this thesis the Nios II have been used, but in the text Nios refer to Nios II core.

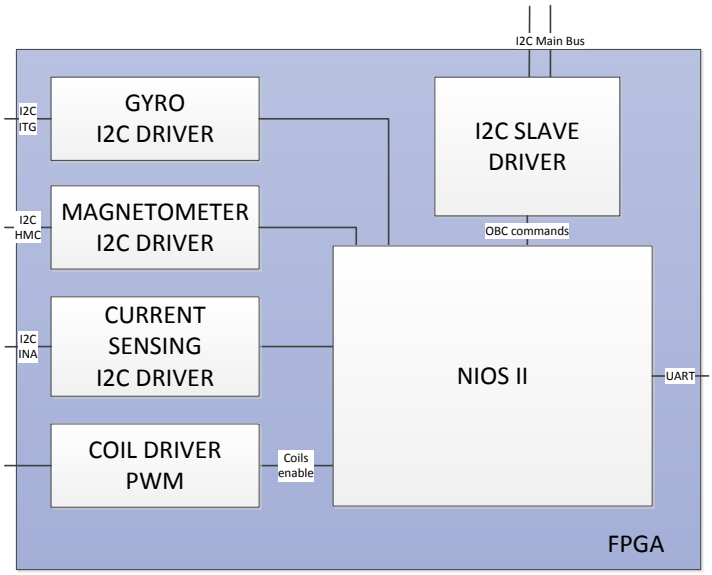


Figure 3.11: System overview over the FPGA on the ADCS card.

Clock

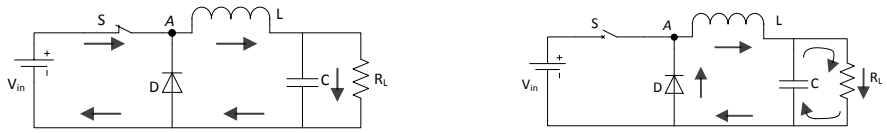
An oscillator with a frequency of 20 MHz was chosen as the system clock for the FPGA. 20 MHz is the lowest frequency possible to use for the Nios core. The power consumption is highly dependent on the frequency. Speed is not a critical factor in a satellite, but the power consumption is. Because of this a low frequency is chosen for the oscillator on the ADCS card.

3.4.6 Power Circuit

On the CubeSTAR satellite an unregulated power bus is used. This means that the distributed power net to all the subsystems will in theory range between 2,5 V and 3,6 V. This causes some considerations for the power distribution on the module card.

In space applications heat dissipation is a problem. Heat dissipation is important to have in mind in every step of the satellites design. On board a spacecraft power consumption is quite critical. For the power regulator on the ADCS card the LM3671 step-down DC-DC switching converters from Texas Instrument are used. The primary advantages of using a switching power regulator instead of a linear regulator, is the high efficiency, smaller size and a lot less heat dissipation.

A switching voltage regulator works on the principle of a switch going on and off at a fixed frequency. The LM3671 IC have three modes of operation PWM,



(a) Closed switch *(b) Open switch*

Figure 3.12: Principle of a switching buck converter

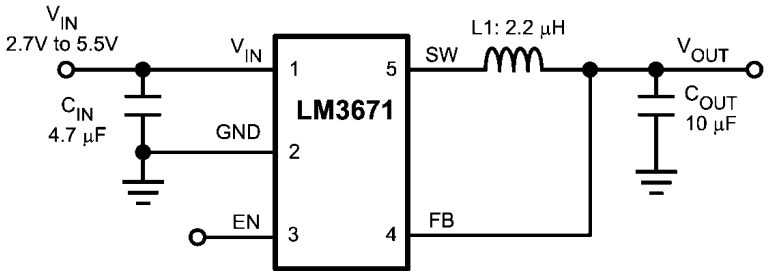


Figure 3.13: The LM3671 voltage converter circuit with external components [17].

PFM (Pulse Frequency Modulation) and shutdown. The mode of operation is dependent on the current required. From approximately 80 mA and higher the IC operates in PWM mode. At lighter load the device automatically switch to PFM mode. This mode causes the IC to operate with a lower switching frequency to maintain a high efficiency.

The PWM mode works on the principle of a transistor connects and disconnect to an inductor. Illustrated by a regular switch in Figure 3.12. The inductor current flows through the capacitor and the load resistance, when the transistor is turned on. The inductor will try to oppose the rising current and an electromagnetic field is generated. When the PWM control turns off the transistor the electromagnetic field in L will start to discharge. The voltage in node A is now forced negative and eventually the diode will be conducting. The current flows through the load and back through the diode. The capacitor discharges into the load during the off time contributing to the total supplied current to the load. [15]

The LM3671 goes in shutdown mode when the voltage is 2,7 V. The theoretically lowest voltage from the power bus is 2,5 V. Because of this it should be considered to give the ADCS a shutdown commando from the OBC before the voltage reaches 2,5 V.

The LM3671 IC only needs a few external components to work properly, see Figure 3.13. Care must be taken when choosing these components. For the inductor there are two main concerns, the inductor must not saturate and the inductor current ripple should be as small as possible. The capacitors should be of ceramic type for low ESR, and X7R/X5R type dielectric for a better performance over temperature. [17]

PCB layout is crucial for getting max performance from a DC-DC switching converter. The component should be placed in a manner so that the switching current loops curl in the same direction. Short and wide tracks are used and a pseudo ground-plane on top of the PCB is connected to the ground plane through vias. The voltage feedback track should be routed away from noisy traces where this is possible. This track could be routed opposite to noisy components in a different layer with a ground plane in between. For minimizing noise to noise sensitive circuits, large distances as possible is recommended.

Chapter 4

Data handling in Nios II

This chapter describes the data handling in Nios and the communication with the OBC.

4.1 Program Flow

All the sensor drivers and the actuator control are written in VHDL. Figure 3.11 illustrates this. To set up the Nios II core and peripherials the program QSYS from Altera is used. Altera Avalon drivers is used for all the components utilized in this project. All the data from the sensors are sent to Nios on different PIOs (Peripheral Input/Output). Each sensor has its own set of PIOs for communication with Nios.

The program flow developed for the Nios is divided into two sections. One part handles the commands from the OBC and the other part is used for magnetorquer control, debuging and printing of sensor data.

Communication with the OBC

The ADCS card functions as a slave on the main bus in the satellite. The OBC is the master of this bus, and the master of all the subsystems in the CubeSTAR satellite. The OBC handles the data sent to and from the ground station. At a later stage in the CubeSTAR project, fully functioning determination and control algorithms will be implemented. For communication to the OBC an I2C slave driver is written in VHDL. This driver controls the communication between the OBC and the ADCS card following the I2C protocol. This driver sends the command number to the Nios core. The number of bytes to respond to the OBC are fixed in the I2C slave driver. This command number is the essence in the instruction handler. The Instruction handler reads the commands and send a given response back to the slave driver and OBC.

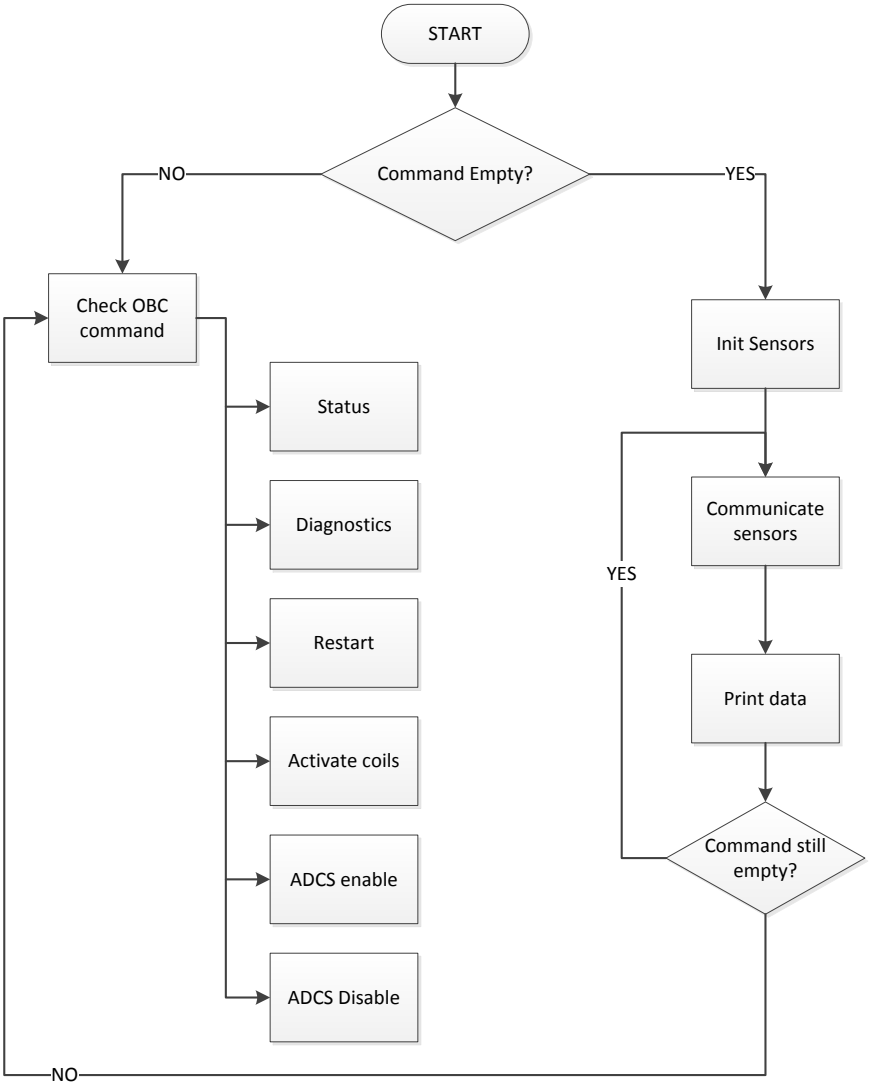


Figure 4.1: A principal drawing of the program flow in the Nios core. The program is built around an instruction handler checking the commands sent from the OBC. If no commands are sent, a printing and sensor debug mode is active.

Table 4.1: Possible Commands sent to the ADCS

Register	Command	Response
ADCS status	0x1E	1 Byte
ADCS diagnostic	0x2D	Sensor values, 57 bytes
ADCS restart	0x55	1 byte
ADCS activate coils	0x87	1 byte
- coil X	0x87	
- coil Y	0x4B	
- coil Z	0x2D	
ADCS enable	0x99	1 byte
ADCS disable	0xAA	1 byte
ADCS get attitude	0xB4	1 byte

Table 4.2: Example on how to activate the Y coil from the ground station.

ADCS address	activate coil command	coil
0x10	0x57	0x4B

More about the I2C protocol is found in section 3.4.1. The commands implemented in the ADCS card are given in Table 4.1.

A command for activating a magnetic coil from the OBC is implemented. This is a backup system if the attitude control or the entire attitude determination and control system fails. A coil can be turned on from the ground station if this happens. Only one coil can be turned on at once. Three bytes are required to activate a coil. The ADCS adress is followed by a write bit, then the activate coil command follows by a command that tells wich coils to be turned on.

Chapter 5

Gyroscope and Magnetometer Calibration

To be able to use the data from the sensors it is desired to know the sensors performance. How large are the error sources and is it possible to correct some of them? In an attitude determination process it is cruical to know if your sensor data is valid. The accuracy of the sensor will affect the accuracy of the final attitude determination and control process. The highest possible accuracy in our sensors are wanted, since the determination and control algorithm contributes to noise or error sources in the final accuracy. Calibration of sensors for attitude determination is often done on ground for a pre-calibration and calibrated in-flight [3].

The pre-flight calibration processes used in this thesis is developed by K. Rensel. This chapter describes the modeling of the different sensors and the method for correcting errors in the measurements. All programs mentioned below are modified slightly to fit this projects specific setup.

5.1 Gyro

The calibration method in this section is made by K. Rensel [21] and based on the work of J.K. Bekkeng [2] [3].

5.1.1 Error Modeling

According to [2] the most significant errors of a MEMS gyro sensor are:

- Scale factor (λ)
- Random bias (η)

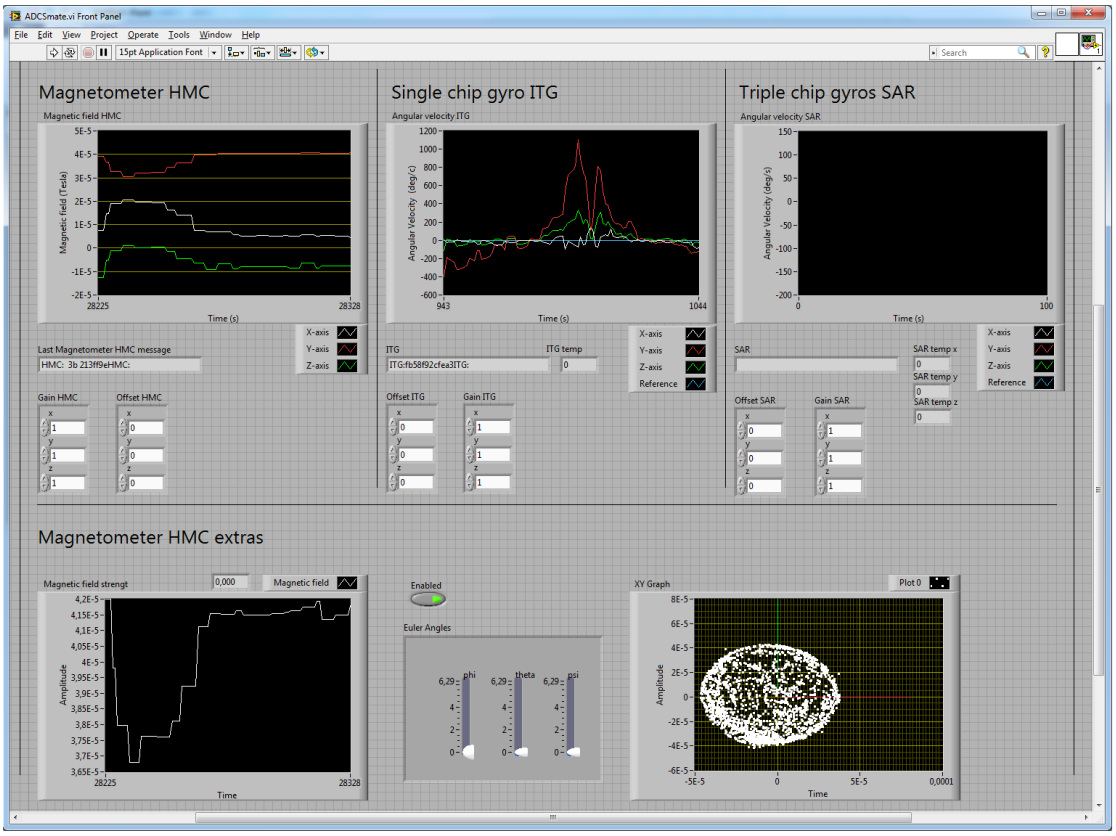


Figure 5.1: The LabView GUI for the calibration of gyro and magnetometer

- Misalignment (δ)
- Temperature dependent bias (b(T))

The signal from the MEMS gyro is modeled:

$$z = S^{-1}(\omega + b + v) \tag{5.1}$$

where the z is the gyros output, S is the scale and misalignment matrix, ω is the true angular rate, b is the bias and v is the gaussian white noise. The above equation is stating that the bias is a stationary value, this is not correct if we look at the temperature dependency. The bias b can be modeled as a sum of a stationary bias, b_0 and a temperaturedependent bias, $b(T)$:

$$b(T) = b_0 + b(T) \tag{5.2}$$

As shown in [21] the gyro is very temperature dependent. This is not taken into account here at this stage.

A misalignment and scale error matrix, S is defined:

$$S = (I + M) = \begin{bmatrix} 1 + \lambda_x & \delta_{xy} & \delta_{xz} \\ \delta_{yx} & 1 + \lambda_y & \delta_{yz} \\ \delta_{zx} & \delta_{zy} & 1 + \lambda_z \end{bmatrix} \tag{5.3}$$

Where M is the misalignment matrix, δ_{ij} is representing the projection of the sensitive axis i on the body axis j. The sensitivity axis x, y and z are intended to be in the same direction as the coresponding body frame axes, so the misalignment is assumed small. See Figure 5.2. The true angular rate in the body frame, ω is given by the below equation.

$$\omega = S(z - b(T) - v)$$

where the S is the scale and misalignment declared earlier, z is the measured angular rate, the b(T) is the temperature dependent bias and v is the Gaussian noise.

5.1.2 Calibration Setup

To calibrate the gyro a reference value is desired. A reference rate table will give a controlled and known angular rate in one axis. An Ideal Aerosmith 1291BR rate table was used for this calibration setup. The rate table is controlled by a serial interface and the rate table returns the true angular rate applied to the system. A LabView program is used to log data from the sensors and a matlab program

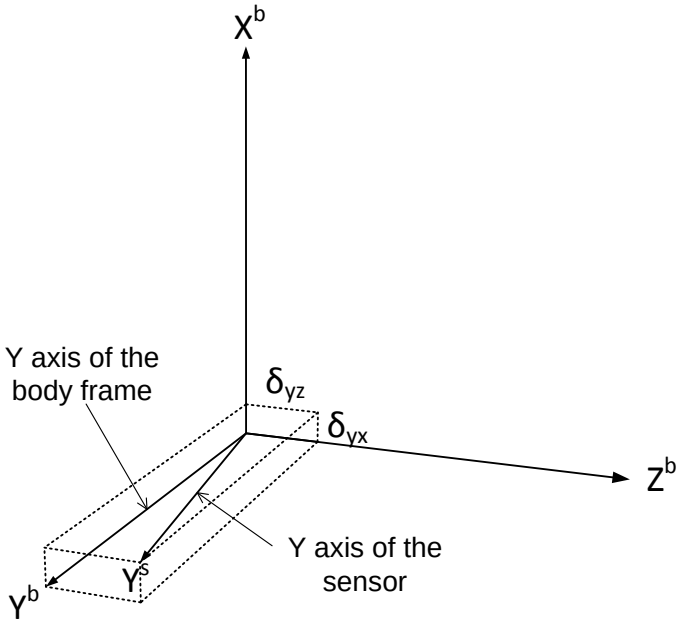


Figure 5.2: Definition of the small misalignment δ, figure by [3].

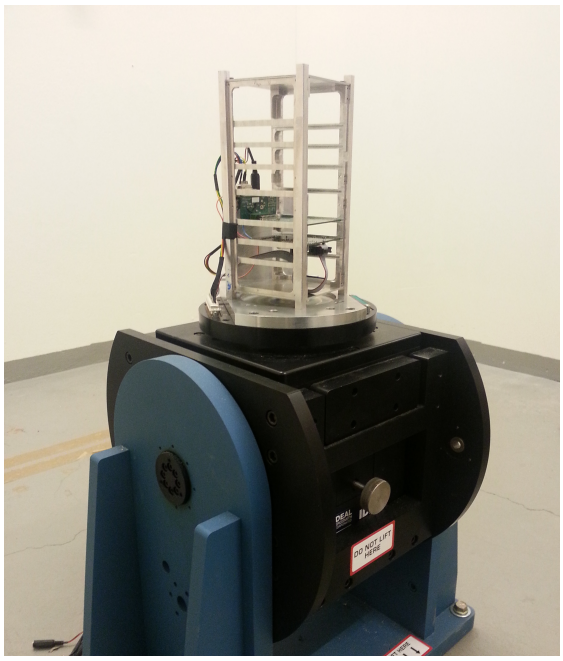


Figure 5.3: The ADCS card with the mini backplane card placed in the satellite structure. The satellite structure can be mounted in different directions, here the satellite is mounted for testing of the z-axis. The ADCS card is connected to the mini backplane card for power and UART communication.

is used for the calibration purpose. Both programs are modified slightly to fit this version of the ADCS card. Both programs are developed with a GUI. The LabView program or Virtual Instrument (VI), controls the rate table and receives data from the ADCS card. The rate table and the ADCS card communicate with the VI through RS-232 serial interface. The VI acquires and plots the sensor data and the reference value at the same time. The ADCS card is placed in a satellite structure mounted on the rate table. The ADCS card is connected to the mini backplane card for RS-232 communication and power. The satellite structure can be mounted in three orthogonal directions for a full testing of all three axes.

5.1.3 Kalman Filtering

A Kalman filter is used to find the optimal parameters for processing the data from the refrence test. A general kalman filter is described more in 2.4.3. The filter utilized here in the calibration process is a simple Kalman filter where the state to be predicted is assumed static [32]. If we assumed the data is temperature compasated, the true angular rate in the body frame of the satellite is given:

$$\omega = Sz - \beta - \eta_v \tag{5.4}$$
$$\omega = (\mathbf{I} + \mathbf{M})z - \beta - \eta_v \tag{5.5}$$
$$\omega = z + \Omega m - \beta - \eta_v \tag{5.6}$$

where

$$\Omega\mathbf{M} = \mathbf{M}z$$

and

$$\Omega = \begin{bmatrix} \omega_y & \omega_z & 0 & 0 & 0 & 0 & \omega_x & 0 & 0 \\ 0 & 0 & \omega_x & \omega_z & 0 & 0 & 0 & \omega_x & 0 \\ 0 & 0 & 0 & 0 & \omega_x & \omega_y & 0 & 0 & \omega_z \end{bmatrix} \tag{5.7}$$

$$m = \begin{bmatrix} \delta_{xy} & \delta_{xz} & \delta_{yx} & \delta_{yz} & \delta_{zx} & \delta_{zy} & \lambda_x & \lambda_y & \lambda_z \end{bmatrix}^T \tag{5.8}$$

The state vector x, is 12 dimensional and consist of 9 scale and misalignment factors and three biases, one for each axis

$$x = \begin{bmatrix} m & \beta \end{bmatrix}^T$$

The filters state equation is estimated as constant, and therefore the state does not change between the update. We can then write the update, or the Kalman state update

$$x_{k+1} = x_k \tag{5.9}$$

A closer look at equation 5.6 and we can find the Kalman measurement equation. We are interested in $\Delta\omega$, the difference in angular rate. The observation matrix, $H = [\Omega - I_{3x3}]$ The measurement equation for our filter is then given

$$\Delta\omega_k = H_k x_k + \eta_k \tag{5.10}$$

where $\Delta\omega_k$ is the measurement at point k, H is the observation matrix of 3x12 elements and v_k is the Gaussian noise. The updated estimate becomes:

$$x = x + K(\Delta\omega - Hx)$$

where K is the Kalman gain and $\Delta\omega$ is the difference between the true angular rate and the measured angular rate.

Matlab Implementation

The whole calibration algorithm is implemented in a matlab program with a GUI interface. The matlab program consists of a pre-processing filter and a Kalman filter. The program only needs data from the spin test as input. The pre-processing is modelled by Equation 5.2 and will remove the temperature dependent bias from the signal. A temperature test has not been executed in this project. The previous results show that the ITG 3200 gyro is highly temperature dependent. The temperature dependent bias is linear and thus easy to compensate for as long as the temperature is known. Because of the lack of temperature coefficient only the stationary bias is removed in this filtering.

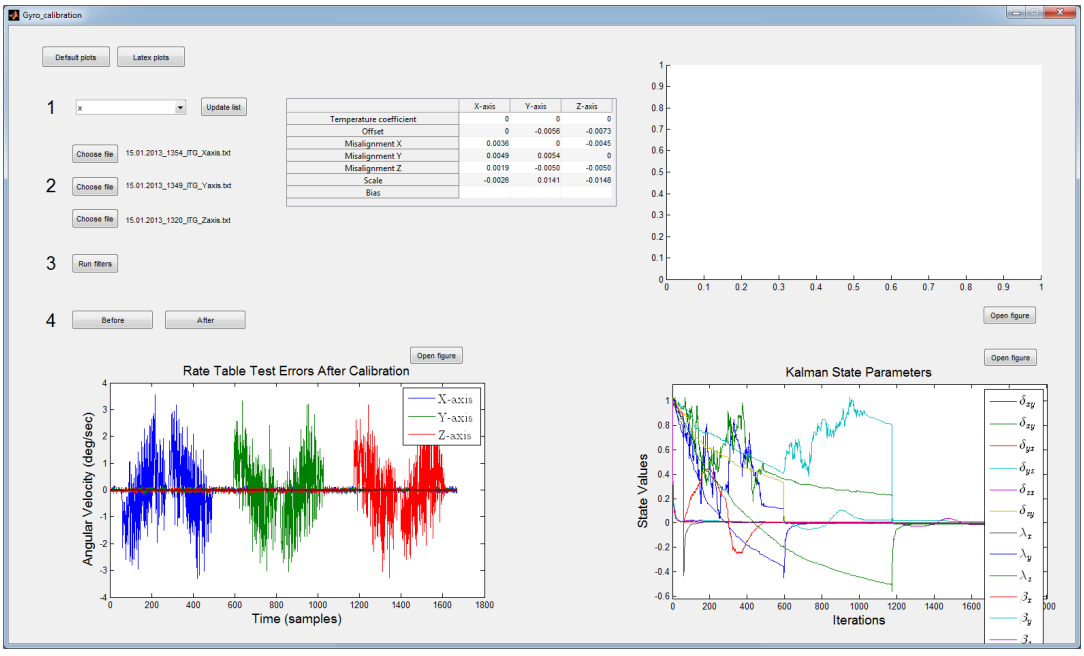


Figure 5.4: The matlab GUI used for the calibration process

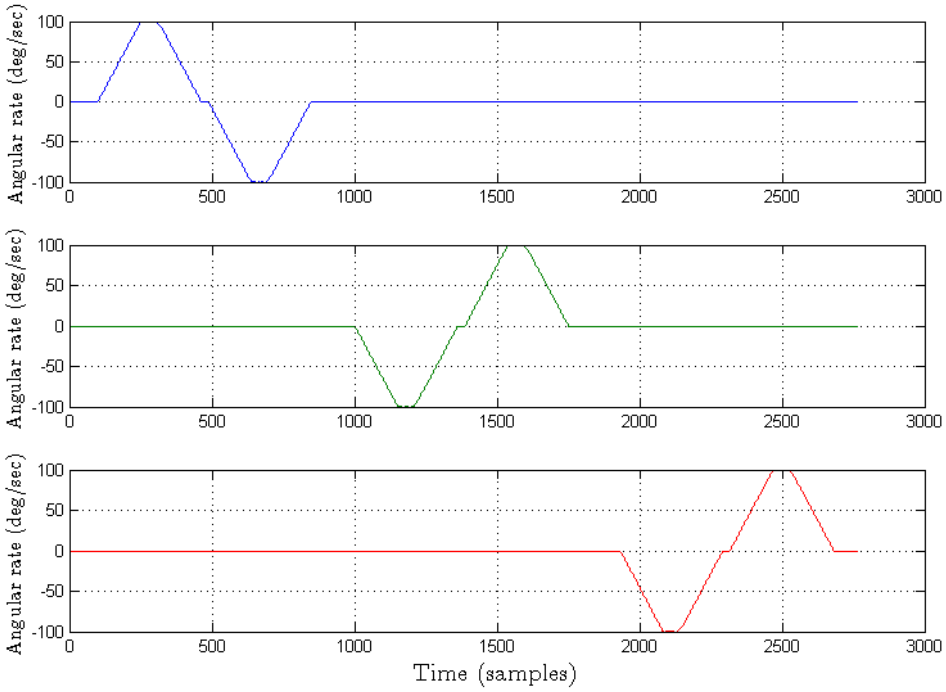


Figure 5.5: The input refrence data to the calibration from the rate table. The separate axes is merged together after the test before calibration is performed.

5.1.4 Results

The spin table test was performed with a maximum rate of 100 deg/sec. The ITG3200 gyro is capable of measuring angular rate op to 2000 deg/sec, but it was not seen as necessary to perform testing on a full scale range. A thesis from CalPoly tech University [19], expected a maximum angular rate of ± 7 deg/sec. Other CubeSats articles estimates an angular rate of only a few deg/sec [26] to ± 20 deg/sec [25]. Once the satellite has been deployed from the launch vehicle an angular rate of a few deg/sec will be present and thus the gyroscope will never measure angular rate at maximum range.

In this section results from the rate table test is shown. The first table shows the parameters for the calibration. In first column the pre-offset bias values are listed, these values are calculated based on the mean of stationary measurements. The rest of the table lists the Kalman state parameters. The results before and after calibration is shown in Figure 5.7 and 5.8.

Table 5.1: Gyro sensor parameters from filtering

Parameter	Symbol	X	Y	Z
Pre offset	b_o	2,280	-0,094	0,7311
Misalignment X	δ_x	0	0,0042	0,0089
Misalignment Y	δ_y	-0,0017	0	-0,0062
Mislignment Z	δ_z	-0,0087	0,0052	0
Scale	S	9,6098e-04	-0,0063	-0,0054
Bias	β	0,0109	-0,0123	0,0040

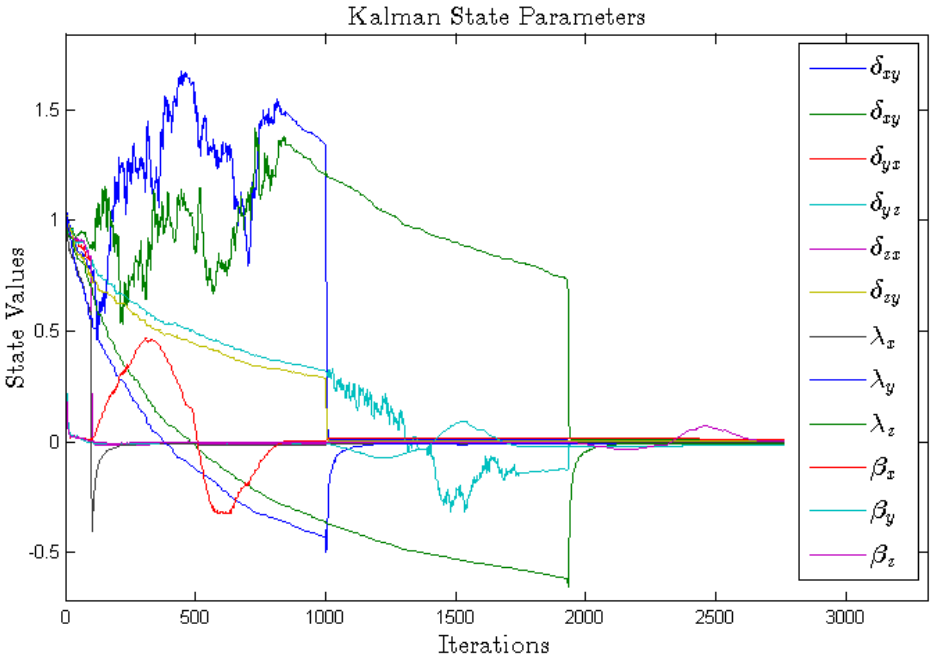


Figure 5.6: The Kalaman state parameters.

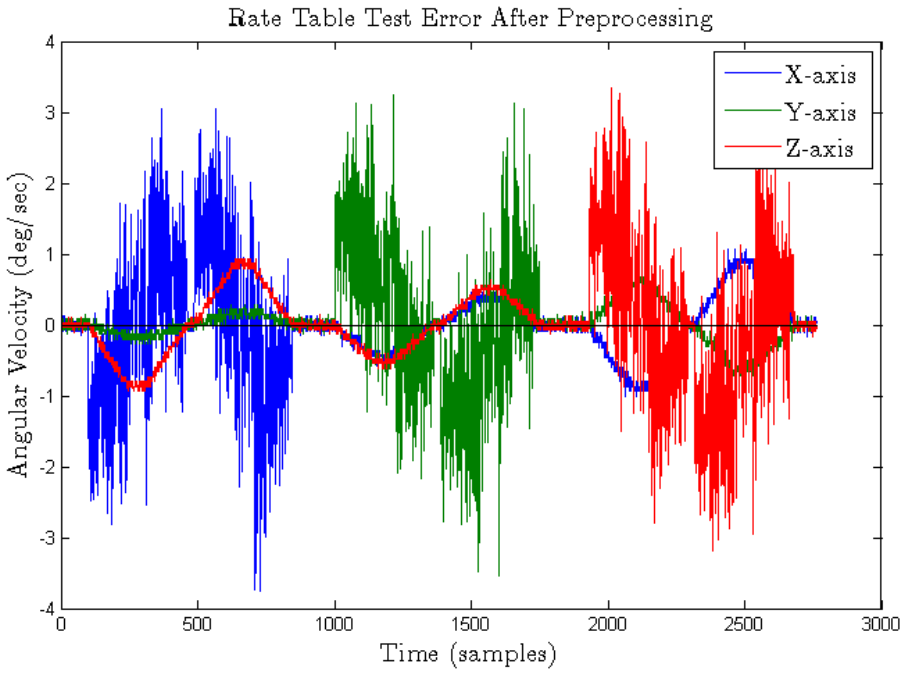


Figure 5.7: The larger offset is removed from the measurements and plotted. Thus there is movement only in one axis an influence is seen in the other axis.

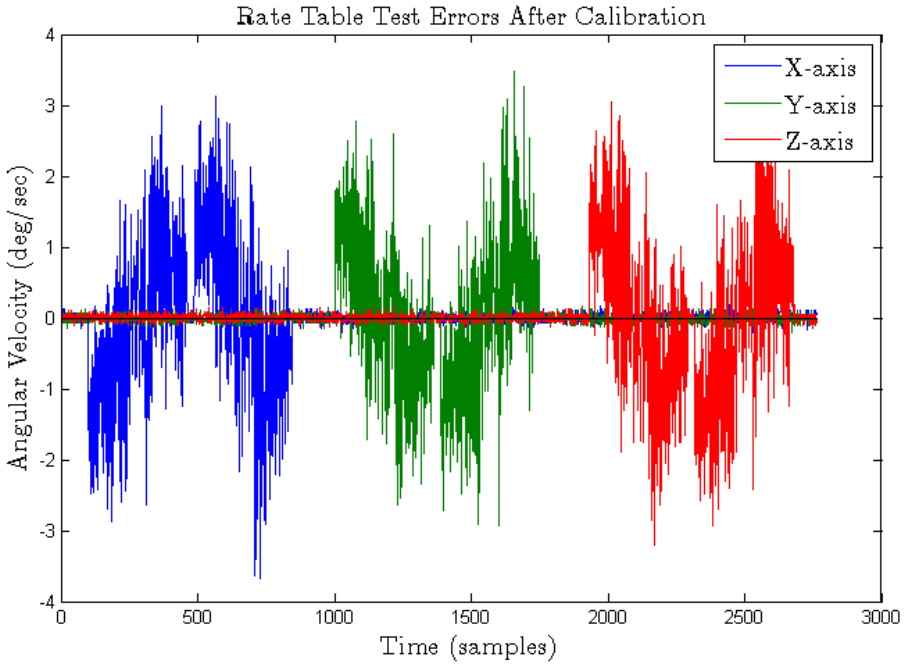


Figure 5.8: The errors shown between the refrence rate table and ITG3200 gyro.

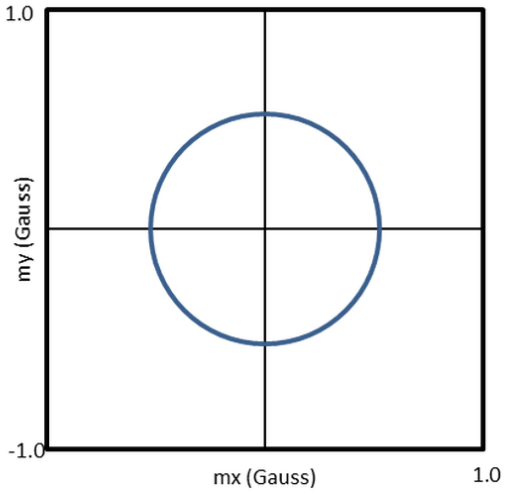


Figure 5.9: A 2D plot from an ideal reading from a magnetometer.

5.2 Magnetometer

5.2.1 Error Modeling

The measurements from the magnetometer will be subjected to different distortions. To get a better accuracy from the magnetometer these sources needs to be investigated and compensated for. [7] describes the output of the magnetometer with five different error sources. The output is corrupted by an additive error of bias and noise. Sensitivity, non-orthogonally and misalignment, together with the sum of soft iron effects corrupt the readings from the magnetometer. The most important error sources are listed and described below.

Hard Iron and Soft Iron

Disturbances of the Earth's magnetic field are a result of external influences to the magnetometer. These disturbances are divided into two categories, hard iron or soft iron. Hard iron disturbances are produced by materials that generate a magnetic field. These are permanent sources to the magnetometer and will generate a constant bias to the magnetometers output. Soft iron disturbances will change the magnetic field. These effects do not come from materials that produce a magnetic field, but from materials that influence the magnetic field. For instance iron and nickel are materials that will generate soft iron disturbances.

These effects can be visualized by a 2D plot of the magnetometer data. The

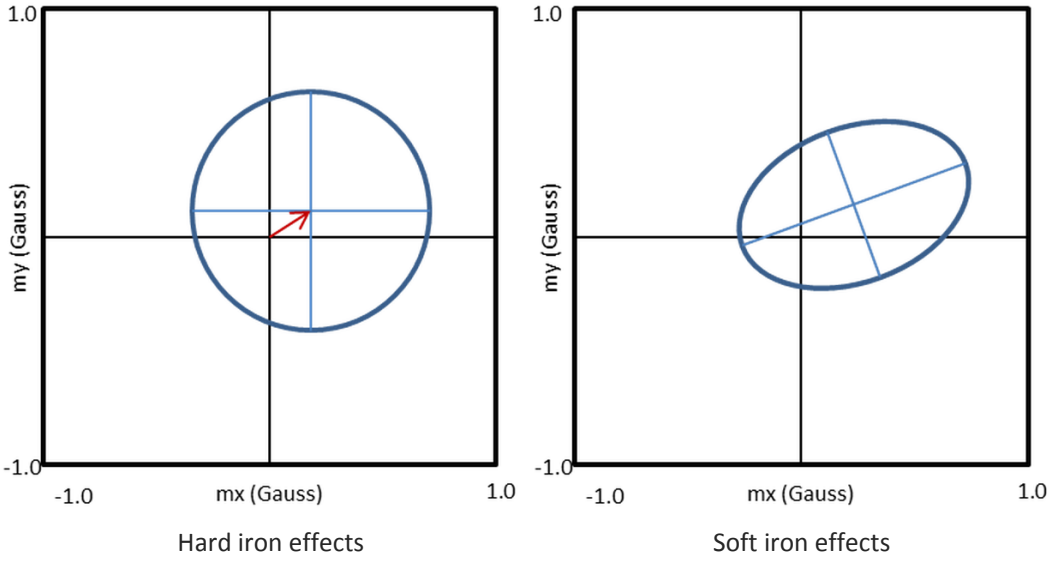


Figure 5.10: Hard iron effects are additive magnetic sources that causes a constant offset. Soft iron causes a change in the magnetic field and the 2D plot will go from an ideal circle to an ellipse. The drawing to the left illustrate the output when both hard iron and soft iron effects are present.

plot from an ideal reading will be a perfect circle centered in (0,0). The magnitude of the magnetic field will be equal to the radius of the circle. Hard iron effects will cause an offset of the center of the circle. Soft iron effects will only distort the existing magnetic field and the output will be an ellipse. When both hard iron and soft iron effects are present the output will be an ellipse with an offset center, as shown in Figure. 5.10. The hard iron error can be denoted as $\mathbf{b}_{HI}$, and the soft iron effect

$$\mathbf{h}_{SI} = \mathbf{C}_{SI}\mathbf{R}^{be}\mathbf{h}^{e} \tag{5.11}$$

Where $\mathbf{C}_{SI}$ is the 3x3 soft iron transformation matrix, $\mathbf{R}^{be}$ the rotational matrix from body to Earth and $\mathbf{h}^{e}$ the magntetic field in Earth (ECEF) frame.

Scaling and bias

Scaling and bias are some of the most common sensor faults and needs to be accounted for. The 3x3 scaling error matrix is denoted $\mathbf{S}_M$ and the bias, $\mathbf{b}_M$.

Misalignment and Non-orthogonality

A non-orthogonality of the sensor can be described by a transformation matrix $\mathbf{C}_{NO}$

$$\mathbf{C}_{NO} = \begin{bmatrix} 1 & 0 & 0 \\ sin(\psi) & cos(\psi) & 0 \\ -sin(-\psi) & cos(\theta)sin(\phi) & cos(\theta)cos(\phi) \end{bmatrix} \tag{5.12}$$

Where (ψ, θ, ϕ) are Yaw, Pitch and Roll seen in the body frame.

Calibration

The output from the magnetometer, $\mathbf{h}_{ri}$ can be descriped by combining all the errors described above:

$$\mathbf{h}_{ri} = \mathbf{S}_M\mathbf{C}_{NO}(\mathbf{C}_{SI}\mathbf{R}_i^{be}\mathbf{h}^{e} + \mathbf{b}_{HI}) + \mathbf{b}_M \tag{5.13}$$

By combining the soft iron error, $\mathbf{C}_{SI}$ with the scaling and non-orthonality matrix, $\mathbf{C}_{SI} = \mathbf{S}_M\mathbf{C}_{NO}\mathbf{C}_{SI}$, the same for hard iron effects, $b = \mathbf{S}_M\mathbf{C}_{NO}\mathbf{b}_{HI} + \mathbf{b}_M$ and setting $\mathbf{h}_i^e = \mathbf{R}_i^{be}\mathbf{h}^{e}$ the output can be described in a more general form:

$$\mathbf{h}_{ri} = \mathbf{C}\mathbf{h}_i^b + \mathbf{b} + \mathbf{n}_{mi} \tag{5.14}$$

where $\mathbf{n}_{mi}$ is the uncorrelated white gaussian niose. According to to [14] a singular value decomposition on $\mathbf{C}$ gives

$$\mathbf{C} = \mathbf{R}_L\mathbf{S}_L\mathbf{V}_L^{'} \tag{5.15}$$

Where $\mathbf{R}_L$ is the rotation matrix, $\mathbf{S}_L$ the scale diagonal matrix and $\mathbf{V}_L$ the orthogonal transformation matrix. L denotes that it is a parameter of an ellipsoid. Both [7] and [14] describes the measurements on the surface on an ellipsoide centered on $\mathbf{b}$. All of the error sources described above will together cause the output from the magnetometer to form a ellipsoide. The goal with any calibration of a magnetometer is therefore to correct the ellipsoide to form a perfect sphere.

[14] suggest a least squares approach to find the ellipsoid parameters, Rotation matrix $\mathbf{R}_L$, bias $\mathbf{b}$ and the scaling matrix $\mathbf{S}_L$. By minimizing the problem

$$\frac{min}{T}\sum_{i=1}^{n}(||\mathbf{T}(\mathbf{h}_{ri}-\mathbf{b}_T)||-1)^2 \tag{5.16}$$

When the optimal values of $\mathbf{T}^*$ and $\mathbf{b}*_T$ is achieved. The singular value decomposition gives

$$\mathbf{T}* = \mathbf{V}_L\mathbf{S}_L^{-1}\mathbf{R}_L^{'} \tag{5.17}$$

The calibrated values are given

$$\mathbf{h}_i^c = \mathbf{S}_L^{-1}\mathbf{R}_L^{'}(\mathbf{h}_{ri}-\mathbf{b}) \tag{5.18}$$

The calibration algorithm used utilizes a brute force method that finds the partial derivative values for $\mathbf{T}$ and $\mathbf{b}$ and use these to solve eq. 5.16. Several methods for solving this problem are given in [14] the article this procedure is based on.

The calibration routine developed in matlab is described belov. From K. Rensel [21]:

1. *Find the beste possible values for $\boldsymbol{T}$ and $\mathbf{b}$, so the process wont take to long. This is performed by looking at det uncalibrated data plot*

2. *Determine the partial derivative for $\boldsymbol{T}$ and $\mathbf{b}$ in turns, based on the derived formula given in [14]:*

$$\nabla\int|T = \sum 2c_t\cdot\boldsymbol{u}_i\otimes\boldsymbol{T}\boldsymbol{u}_i \tag{5.19}$$

$$\nabla\int|b = \sum -2c_t\cdot\boldsymbol{T}'\boldsymbol{T}\boldsymbol{u}_i \tag{5.20}$$

   where $\boldsymbol{u}\equiv\boldsymbol{h}_{ri}-\boldsymbol{b}$, and $c_T = 1-||\boldsymbol{T}\boldsymbol{U}_i||^{-1}$

3. *Correct $\boldsymbol{T}$ and $\mathbf{b}$ by a small fraction of the corresponding partial derivative.*

4. *Calculate the error by Equation 5.16.*

5. *Check if the step was successful (less error than last time), and adjust the fraction from 3, up if the error was decreasing and down if not.*

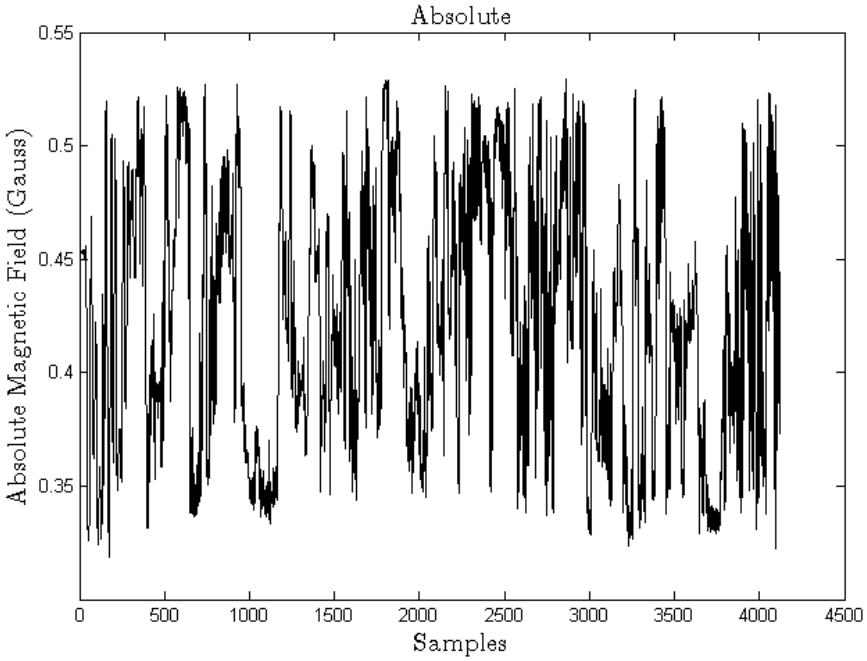


Figure 5.11: The absolute values of the magnetic field recored. Uncalibrated data

5.2.2 Results

The calibration of the magnetometer was done by turning the ADCS card in arbitrary directions. This was done while plotting the data in a 2D plot with LabView. By doing this, it was easy to see if a sufficient distribution of plots where recorded.

Discussion

The HMC5883 magnetometer has been calibrated using the algorithm developed described above. This is a simple, brute force method for finding the ellipsoid parameters. The magnetometer data is corrected to a circle. This is not an accurate calibration since there is no knowledge about how strong the magnetic field is.

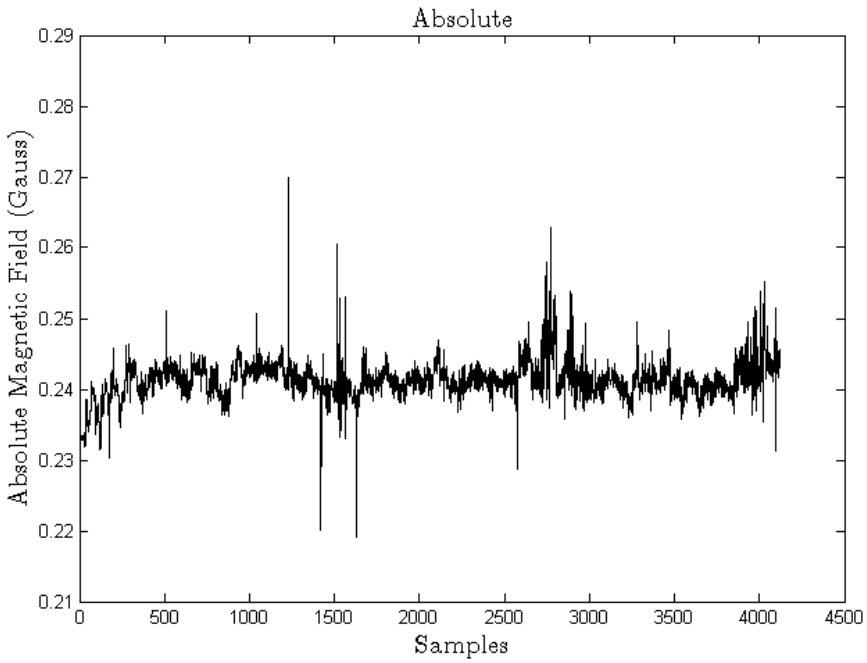


Figure 5.12: The absolute values measured by the HMC5883 magnetometer.

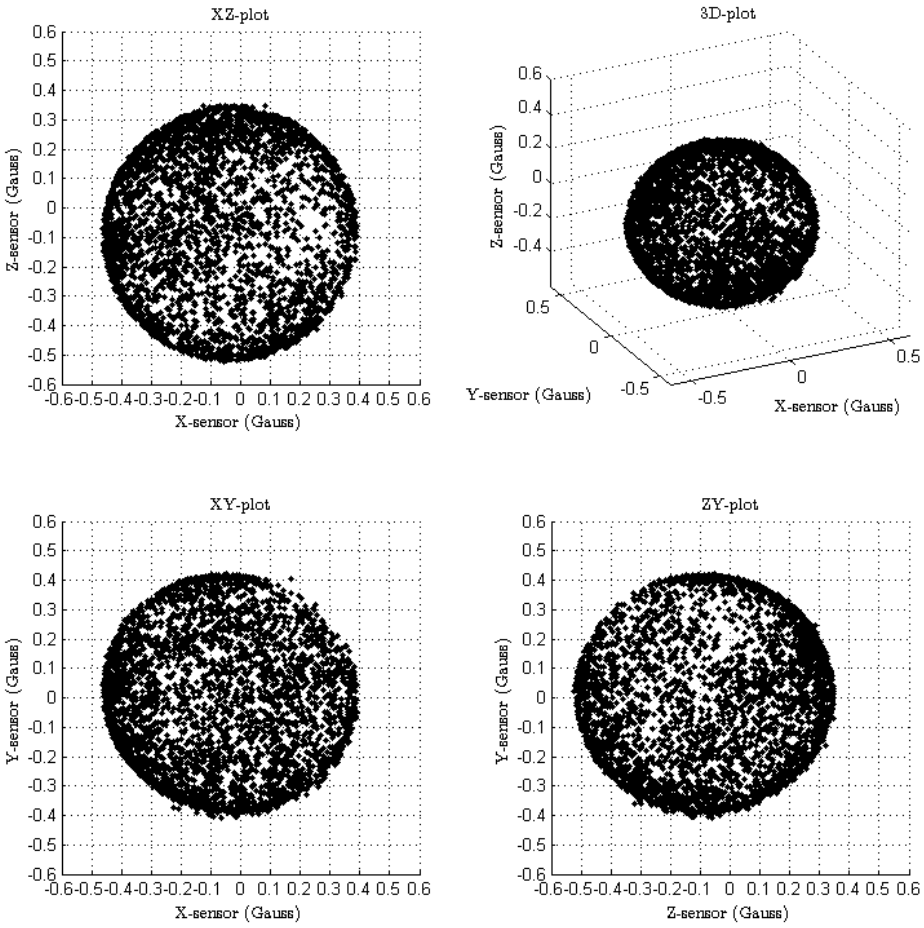


Figure 5.13: Uncalibrated values from the HMC883 magnetometer. An offset is seen. The dataset is weakly elliptical.

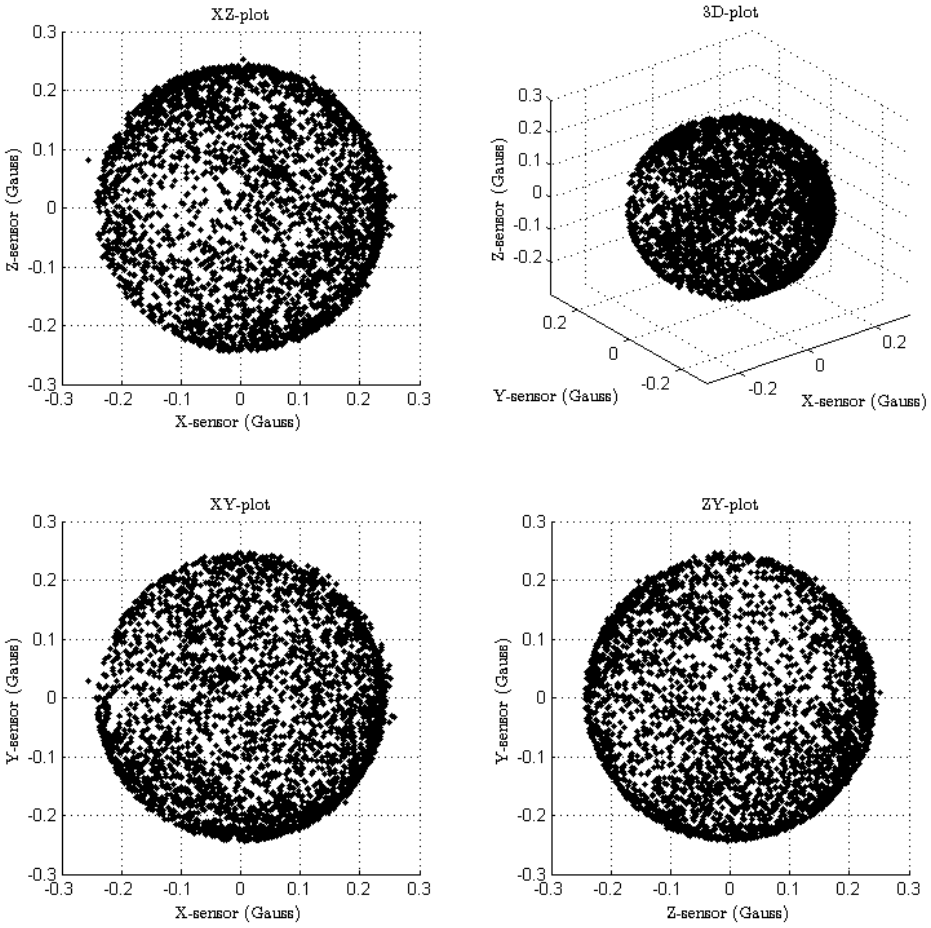


Figure 5.14: Calibrated values from the HMC883 magnetometer. The data set is now more psherical and centered.

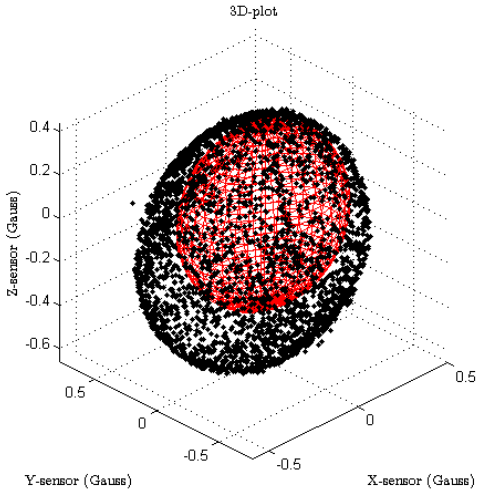


Figure 5.15: The uncalibrated magnetometer data is plotted with a perfect sphere for comparison. It's clear to see that the magnetometer data pocess an offset, but it's only minor elliptic.

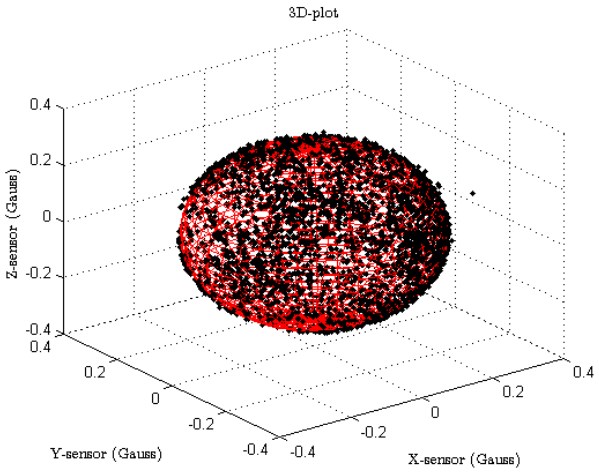


Figure 5.16: The calibrated magnetometer data is plotted with a perfect sphere for comparison. The dataset fit around the edge of the sphere.

Chapter 6

Magnetorquer Testing and Filter Implementation

This chapter describes the testing of the magnetorquers and coil drivers. A Kalman filter has been implemented on the FPGA for demonstration purpose for the ADCS card. Results from implementation of the voltage converters are described in the last section in this chapter.

6.1 Magnetourquer Testing

CubeSTAR is planned to fly with three magnetic coils, two coils on the long sides, x and y coils. The shorter coil is located at the top of the satellite, named the z coil. The shorter coil has been tested with the ADCS card by regulating the current in steps. The long side coil, x or y coil is tested with maximum current.

Magnetic Coils

A short magnetic coil has been produced with the coil winder machine. The coil frame measure 83x84 mm and are placed between all the poles on top of the CubeSTAR satellite. The coil frame is made out of Teflon plastic and printed by a 3D printer at the mechanical workshop. The coil was produced with 252 turns of 0,15 mm copper wire. The coil should be considered formed as a rectangle for a better utilization of the top area. A larger face area will increase the magnetic moment produced by the coil. It is very important that there is proper isolation between the copper wire and the conduction aluminium frame. It may be considered to reduce the number of turns some because of this. The long coil tested was previously produced with 269 turns of copper wire.

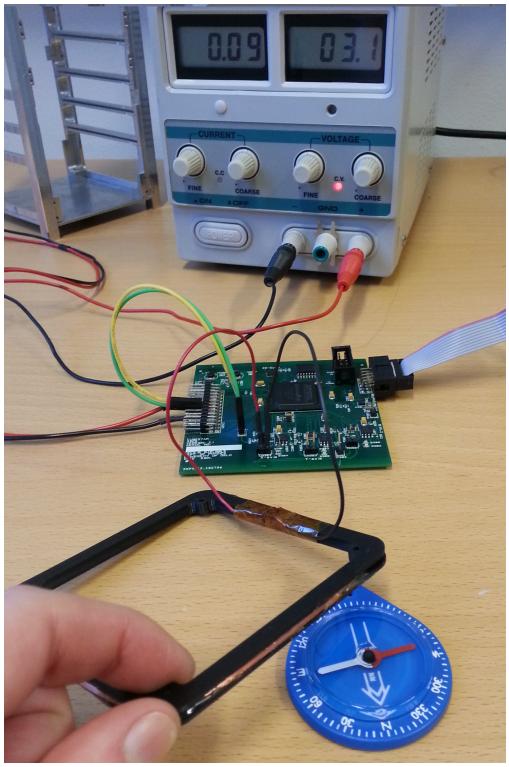


Figure 6.1: Testing of a small coil with the ADCS card. A compass needle is used to verify the direction of the current. By changing the current direction the compass needle turns in opposite direction.

Setup

The setup is shown in Figure 6.1. The coil is driven by an H-bridge (see section 3.4.4) which is controlled by a PWM signal generated by the FPGA. The pulse width module is written in VHDL and the direction of the current and the width of the pulse are controlled from Nios. The enable signal sets all of these parameters. The enable signal is a vector of six elements divided into two categories; the first 3 bits determines which coil to be turned on and the direction, the last 3 bits determines the duty cycle of the pwm signal given to the H-bridge. The commands are listed in Table 6.1 and 6.2.

The coil tested is produced with 252 turns of 0,15 mm copper wire. The voltage applied to the coil driver circuit is 3,3 V. The calibration values for the current sensing circuit are calculated based on a maximum current of 110 mA and a shunt resistor of 1,5 Ω.

Table 6.1: The possible coil commands. All commands are in binary.

Coil Command	Coil
000	Break Mode
001	X axis
010	-X axis
011	Y axis
100	-Y axis
101	Z axis
110	-Z axis
111	Break Mode

Table 6.2: Pulse width commands. All commands are in binary.

Width Command	Duty Cycle
000	12,5 %
001	25,0 %
010	37,5 %
011	50,0 %
100	62,5 %
101	75,0 %
110	87,5 %
111	93,8 %

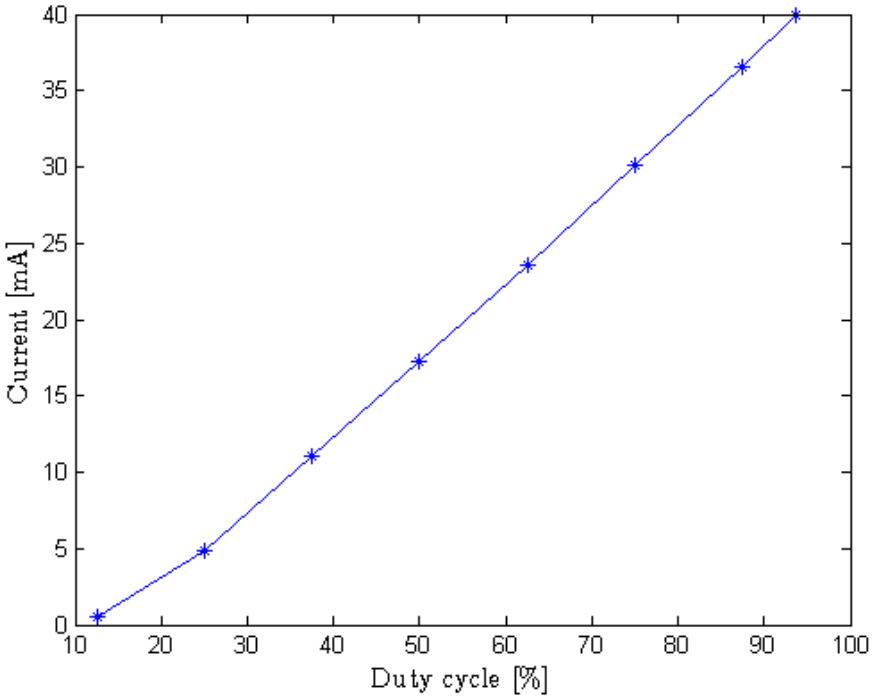


Figure 6.2: The current through a short coil versus the duty cycle of the H-bridge input signal. The powersupply to the H-bridge are set to 3,3 V.

Results

The results from sending current in one direction through the short coil is plotted in Figure 6.2. The duty cycle is changed by changing the command sent from the Nios core. Current measurements are taken with the INA226 current monitor chip. Table 6.2 shows a clear linear dependency between current and duty cycle from 25 % - 93,8 %. The result from 12,5 % duty cycle is so small it is regarded as smaller than the uncertainty of the measuring process.

For verification of the current direction through the coil, a standard compass is used. By changing the direction of the current through the coil, the compass

Table 6.3: Maximum current through short coil, z and long coil, x/y.

Coil	Duty Cycle	Current
Short coil, z	93,8 %	40 mA
Long coil, x/y	93,8 %	20 mA

needle will turn in the opposite direction.

Discussion

[21] developed a spreadsheet for the production of the magnetic coils. [27] set a maximum reference value for the magnetic moment to 100 mAm^2, much lower values than this will be able to control the small satellite CubeSTAR. Equation 2.10 states that the magnetic dipole moment is the product of number of coil turns, current and face area $\mu = nIA$. When a coil is already produced the number of turns, and face area are constant. By knowing the current sent through the coil, the magnetic dipole moment of the coil will be known. The largest possible current sent through the coil was 40 mA when the number of turns are 252 in a small coil and a voltage of 3,3 V is applied to it. This will result in a magnetic dipole moment of 68 mAm^2.

The produced coils have been produced with a core of plastic. The coils launched into space will consist of a space apprved material (POM or PEEK will be used for CubeSTAR). When the final coils are produced the number of turns in the coil will be known. For the final ADCS there is no need for current regulations in the coils. How much current that are wanted in the coils will depend on the final number of turns and the wanted magnetic moment for each coil.

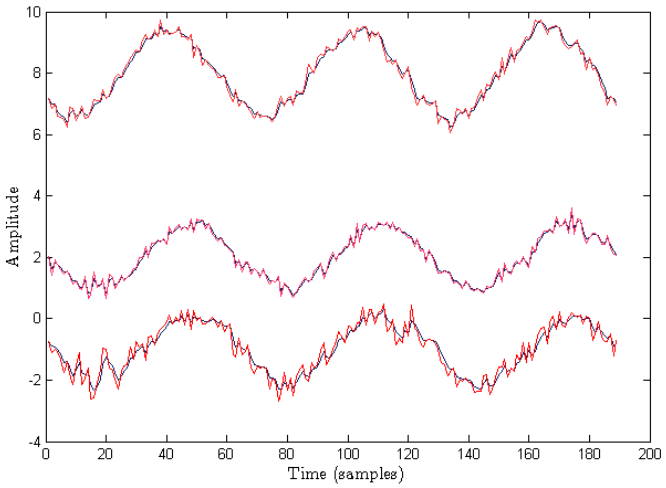


Figure 6.3: Matlab simulation of the Kalman filter. The noisy input signal in red. The filtered signal is shown in black. The outut signal from the filter manage to reduce the noise.

6.2 Filter Implementation

6.2.1 Matlab Simulation

As described in section 2.4.3, the Kalman filter is a common algorithm for attitude determination [32]. In this section the possibilities for implementing a filter like this on the ADCS card is investigated.

A Kalman filter was written in Matlab. This is a simple Kalman filter, filtering a noisy sin signal that simulate an angular rate. The measurement or the input signal, z is given

$$z = C + sin(t) \tag{6.1}$$

where C is a constant given the signal a constant bias and t is the simulated time. White noise is added to the signal.

The filter is simulated in matlab and it manages to filter out the noise. The filtered signal is now almost noise free. The noise can be reduced further by changing the uncertainties in the filter estimate. By doing this, the filter tends to give a larger lag of the signal. T

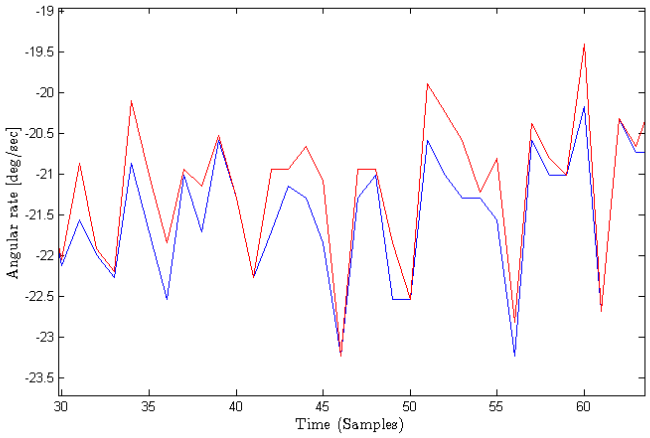


Figure 6.4: Kalman filtering of gyro signal. The raw data from the gyroscope is shown inn red while the filtered signal is shown in blue.

6.2.2 FPGA Implementation

The Kalman filter written in matlab where converted to VHDL with the use of Mathworks HDL Coder. The generated code where simulated using ModelSim. In ModelSim it was verified that the filter manage to filter the simulated signal by looking at the input and output of the filter. The Kalman filter where implemented in the Cyclone IV FPGA for testing of raw data from the gyroscope.

The filter where fed with data from all three axes of the gyroscope. The results from the filter and the raw data where sent to Nios. The raw data and the filtered data where sent to LabView for logging over UART communication from Nios. Static testing where performed. Some of the data is shown in Figure 6.4.

The data plotted in Figure 6.4 shows the raw gyro data and filtered data. The filtered data follows the input data. This filter does not perform very well. The filters noise parameters are not correct and adapted to the raw gyro data. The filters performance where not the main goal of this testing. A tuning of the filters noise parameters would have fixed this problem.

This simple Kalman filter with only three variables to predict is quite simple. Larger more complex state matrix is often seen in this type of filtering. It will still serve as an example if this filter is feasible on the Cyclone IV FPGA. The filter is implemented on the Cyclone IV FPGA and use 148 logic elements and 6 embedded multipliers. For comparison, the Cyclone IV has 114 480 logic elements, the filter uses under 1 % of this.

6.2.3 Control Algorithm

The previous simulink code for the satellite controller is tested on both a Stratix III and a Cyclone IV. The simulink code from Stray was programed to a DE2-115 development kit and hardware in the loop test (HIL) was performed with the use of Simulink. The satellites controller is made in matlab and simulink with the use of DSP builder blocks. With the use of DSP builder, HDL files are generated. By compiling the project from the DSP builder and simulink control-project, Quartus II gives valuable information about how much logic elements and multipliers the algorithm uses on the FPGA.

The control algorithm developed uses 360 9-bits multipliers and approximately 7000 logic elements. The Kalman filter and the control algorithm have been successfully implemented on the Cyclone IV (EP4CE115F23) FPGA.

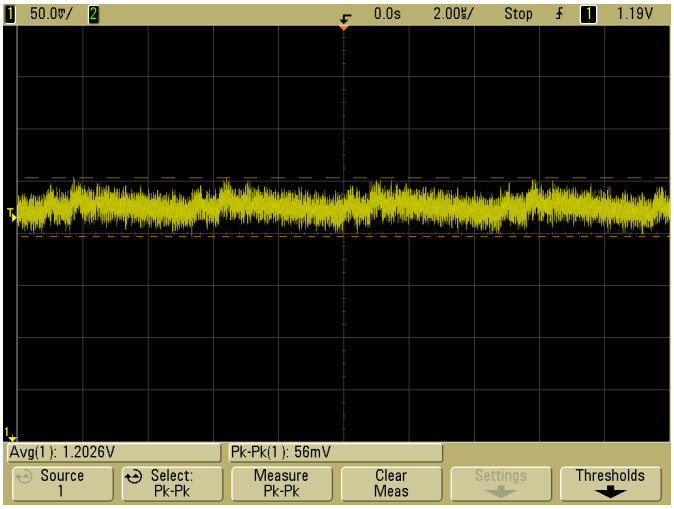


Figure 6.5: The 1,2 V power converter. A peak-peak value of 49 mV

6.3 Voltage Converters

On the ADCS card two voltage converters are implemented. They are both of the type LM3671 fromTexas Instrument converting the voltage down to 1,2 V and 2,5 V. A maximum of 50 mV p-p is stated in the datasheet [17]. The screenshot from an oscilloscope verifies this. Both of the converters is function as stated from the manufacturer. The Figures 6.5 and 6.6 gives a peak-peak value of 49 mV and 56 mV.

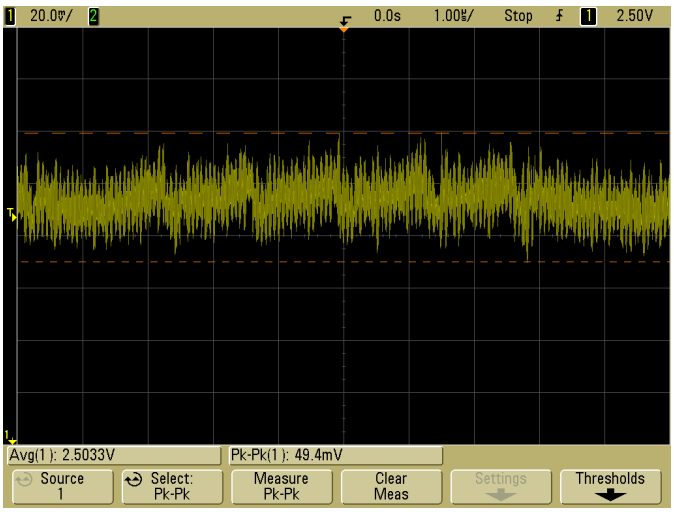


Figure 6.6: The 2,5 V power converter. A peak-peak value of 56 mV

Chapter 7

Summary and Future Work

7.1 Sumamary of the Present Work

The work of this thesis has been related to the further development of the ADCS on CubeSTAR. This thesis describes the design, development and analysis of the hardware for an ADCS.

A new version of the ADCS card has been made with all the hardware necessary for implementation of the complete ADCS on CubeSTAR. A new hardware platform for the ADCS card is chosen, a Cyclone IV FPGA. The FPGA chosen will have the necessary computational power to implement the Attitude Determination and Control algorithm at a later stage in the project. A Nios II core is implemented for the data handling on the FPGA. A gyroscope and magnetometer are implemented and the interface for the sun sensor is in place.

Coil drivers have been produced with a new current sensing circuit. A new coil for the short side, the z-axis has been produced. This is tested together with the coil driver. The coil driver successfully manages to send current in both directions, regulate the current and perform current measurements.

Calibration routines have been performed on both the gyroscope and magnetometer. An ellipsoide fitting calibration is performed on the magnetometer.

The ADCS card works as a slave for the OBC on the I2C bus. Interface to the OBC is in place. Testing between the two module card have been performed and data have successfully been sent between them.
A Kalman filter is implemented as a demonstration of a possible filter for the attitude determination part.

7.2 Future Work

Based on the experience from this work the following subjects should be part of the future work:

The voltage converters implemented performed good and as specified by the manufacturer. However, the 2,5 V converter broke down two times during this project. The reason for this is not known, one possible reason for this may be a too high capacitive load for the converter. This should be looked further into and fully tested.

InvenSense has released a new MEMS gyroscope on the market. This new model, ITG3050 has an adjustable maximum range. This will increase the resolution from 0,07 $(deg/sec)/bit$ from the implemented gyro to 0,008 $(deg/sec)/bit$ with a maximum range of $\pm$ 250 deg/sec for the new gyro model. The new gyro, ITG3050 is easy to swap with the existing gyro since the same footprint has been used for both models. The magnetometer should be calibrated with an accurate reference.

The sun sensor should be implemented on the ADCS card and tested. An in-flight calibration filter should be developed for the gyro and magnetometer together with a full attitude determination algorithm. The Attitude Determination and Control algorithm should be developed together with the hardware limitations in mind. The whole system should be tested thorough and hardware in the loop simulation performed.

The possibilities for using a smaller FPGA should be investigated. The Cyclone IV package chosen consist of a large number of logic elements. This decision should be made when there is more knowledge about the determination algorithm for CubeSTAR.

Communication to the OBC is tested with the OBC module card. Full testing should be done with both module cards connected to the back plane.

New coil frames must be produced in the proper material approved for space. The desired coil current should be determined for the x, y coils and the z coil. Thorough system testing should be performed with the coils mounted on the side panels and the ADCS card mounted on the back plane.

Bibliography

[1] *Application Note AN213*. Tech. rep. www.magneticsensors.com. Honeywell.

[2] Jan Kenneth Bekkeng. "Calibration of a Novel MEMS Inertial Reference Unit". In: *IEEE Transactions on Instrumentation and Measurement* 58.6 (Nov. 2007), pp. 1967–1974.

[3] Jan Kenneth Bekkeng. "Prototype Development of a Low-Cost Sounding Rocket Attitude Determination System and an Electric field Instrument". PhD thesis. University of Oslo, Jan. 2007.

[4] T. A. Bekkeng et al. "Design of a multi-needle Langmuir probe system". In: *Measurement science and technology* 8.2 (July 2010).

[5] Tore André Bekkeng. "Prototype Development of a Multi-Needle Langmuir Probe System". MSc thesis. University of Oslo, 2009.

[6] *Cyclone IV Device Handbook Volume 1*. www.altera.com. Altera.

[7] Jiancheng Fang et al. "A Novel Calibration Method of Magnetic Compass Based on Ellipsoid Fitting". In: *IEEE Transactions on Instrumentation and Measurement* 60.6 (June 2011).

[8] Jacob Fraden. *Handbook of Modern Sensors*. 4th edition. ISBN 978-1-4419-6465-6. Springer, 2010.

[9] Agnar Grødal. *Elektromagnetisk kompabilitet for konstruktører*. (in Norwegian). Tapir Forlag, 1997.

[10] Markus A Grønstad. "Implementation of a communication protocol for CubeSTAR". MSc thesis. University of Oslo, 2010.

[11] *H-bridge Drivers*. BD6210 www.rohm.com. ROHM Semiconductor.

[12] Tai-Ran Hsu. *MEMS and Microsystems- Design, Manufacture and Nanoscale Engineering*. 2th edition. ISBN 978-0-470-08301-7. Wiley, 2008.

[13] *ITG-3200 Product Specification*. 1.4. www.invensense.com. InvenSense Inc. Mar. 2010.

[14] Vasconcelos J.F. et al. "Geometric Approach to Strapdown Magnetometer Calibration in Sensor Frame". In: *Aerospace and Electronic Systems, IEEE Transactions on* 47.2 (Apr. 2011). ISSN: 0018-9251.

[15] T. F. Bogart Jr, J. S. Beasley, and g. Rico. *Electronic Devices and Circuits*. 6th edition. ISBN 0-13-111142-6. Pearson, 1997.

[16] Rudolph Emil Kalman. "A New Approach to Linear Filtering and Prediction Problems". In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D (1960), pp. 35–45.

[17] *LM3671, 600mA Step-Down DC-DC Converter*. Tech. rep. www.ti.com. Texas Instruments.

[18] *Magnetic Sensor Overview*. Tech. rep. www.magneticsensors.com. Honeywell.

[19] Matthew Thomas Nehrenz. "Initial Design and Simulation of the Attitude Determination and Control System for LightSail-1". BSc thesis. California Polytechnic State University, 2010.

[20] Martin Oredsson. "Electrical power system for the CubeSTAR nanosatellite". MSc thesis. University of Oslo, 2010.

[21] Kjetil Rensel. "An Attitude Detumbling System for the CubeSTAR Nano Satellite". MSc thesis. University of Oslo, Aug. 2011.

[22] Cal Poly SLO. "The CubeSat program. CubeSat Design Specification rev. 12". http://www.cubesat.org/index.php/documents/developers. 2009.

[23] Joe Seeger, Martin Lim, and Steve Nasiri. *Development of High-Performance High-Volume Consumer MEMS Gyroscopes*. http://www.invensense.com. InvenSense Inc. 1197 Borregas Ave, Sunnyvale, CA 94089 U.S.A.

[24] Marcel J Sidi. *Spacecraft Dynamics and Control - A practical engineering approach*. ISBN 978-0-521-55072-7. Cambridge University Press, 1997.

[25] John C. Springmann, Benjamin P. Kempke, and James W. Cutler. "Initial Flight Results of the RAX-2 Satellite". In: (). 26th Annual AIAA/USU Conference on Small Satellites.

[26] Willem H. Steyn and Vaios Lappas. "Cubesat solar sail 3-axis stabilization using panel translation and magnetic torquing". In: *Aerospace Science and Technology* 15.6 (2011), pp. 476 –485. ISSN: 1270-9638.

[27] Fredrik Stray. "Attitude Control of a Nano Satellite". MSc thesis. University of Oslo, Oct. 2010.

[28] *Three-Axis Digital Compass IC HMC5883L*. www.magneticsensors.com. Honeywell.

[29] Johan Ludvig Tresvig. "Design of a prototype communication system for the CubeSTAR nano-satellite". MSc thesis. University of Oslo, 2010.

[30] Henning Vangli. "Construction of a remotely operated satellite ground station for low earth orbit communication". MSc thesis. University of Oslo, 2010.

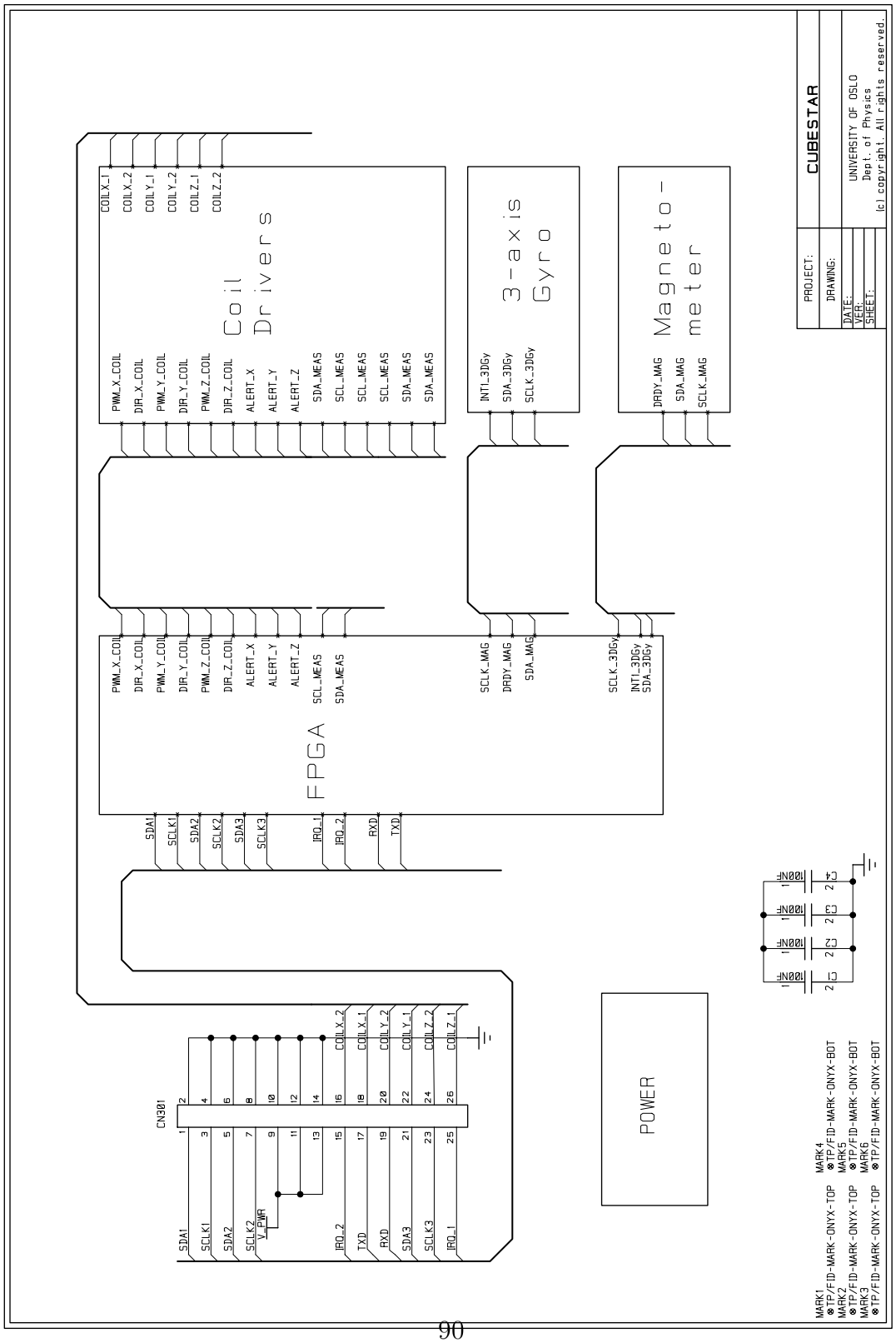
[31] Erik Vikan. "Electrical power system for the CubeSTAR nanosatellite". MSc thesis. University of Oslo, 2010.

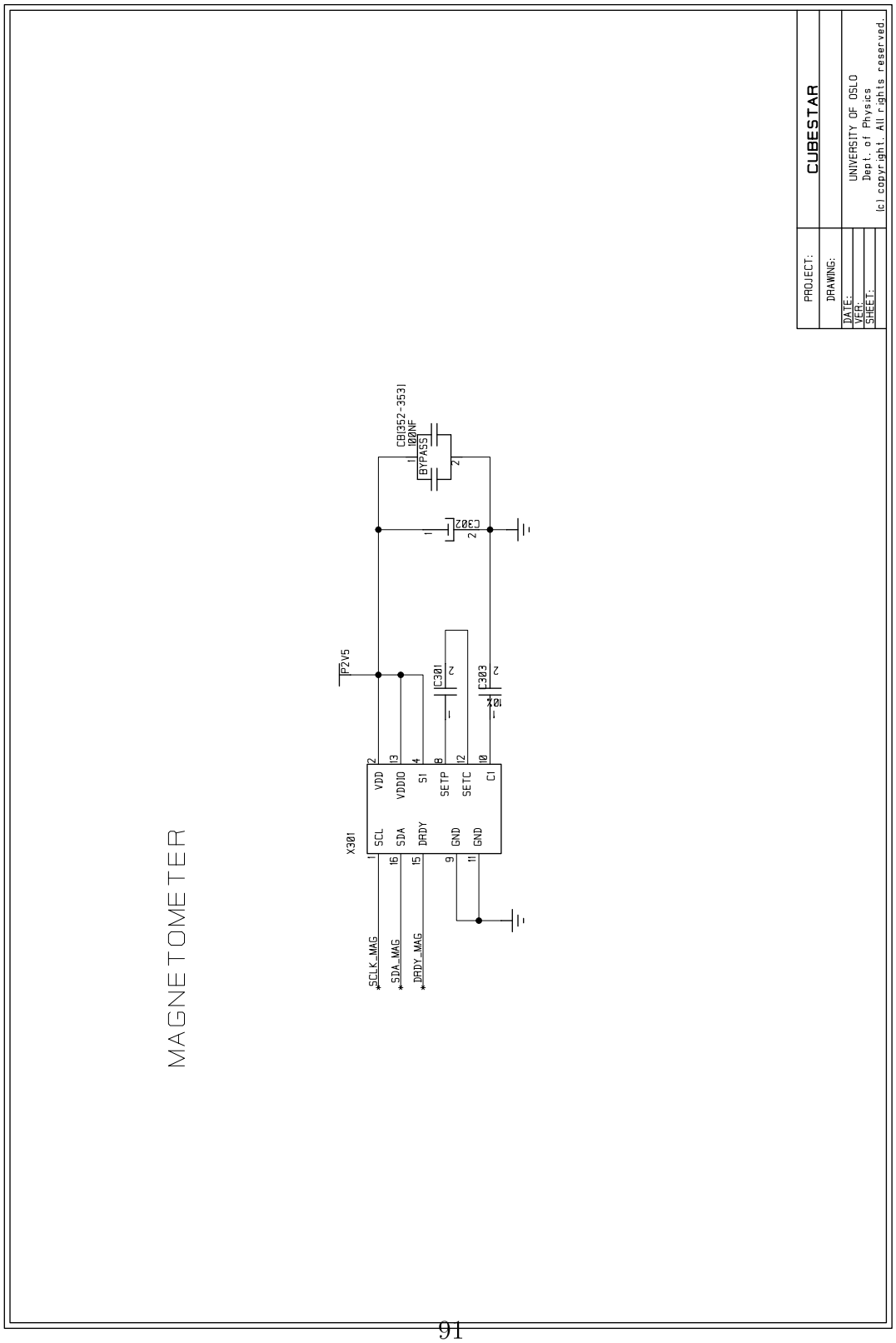
[32] James R. Wertz, ed. *Spacecraft Attitude Determination and Control*. Vol. 73. ISBN 90-277-1204-2. Kluwer Academic Publishers, 1978.

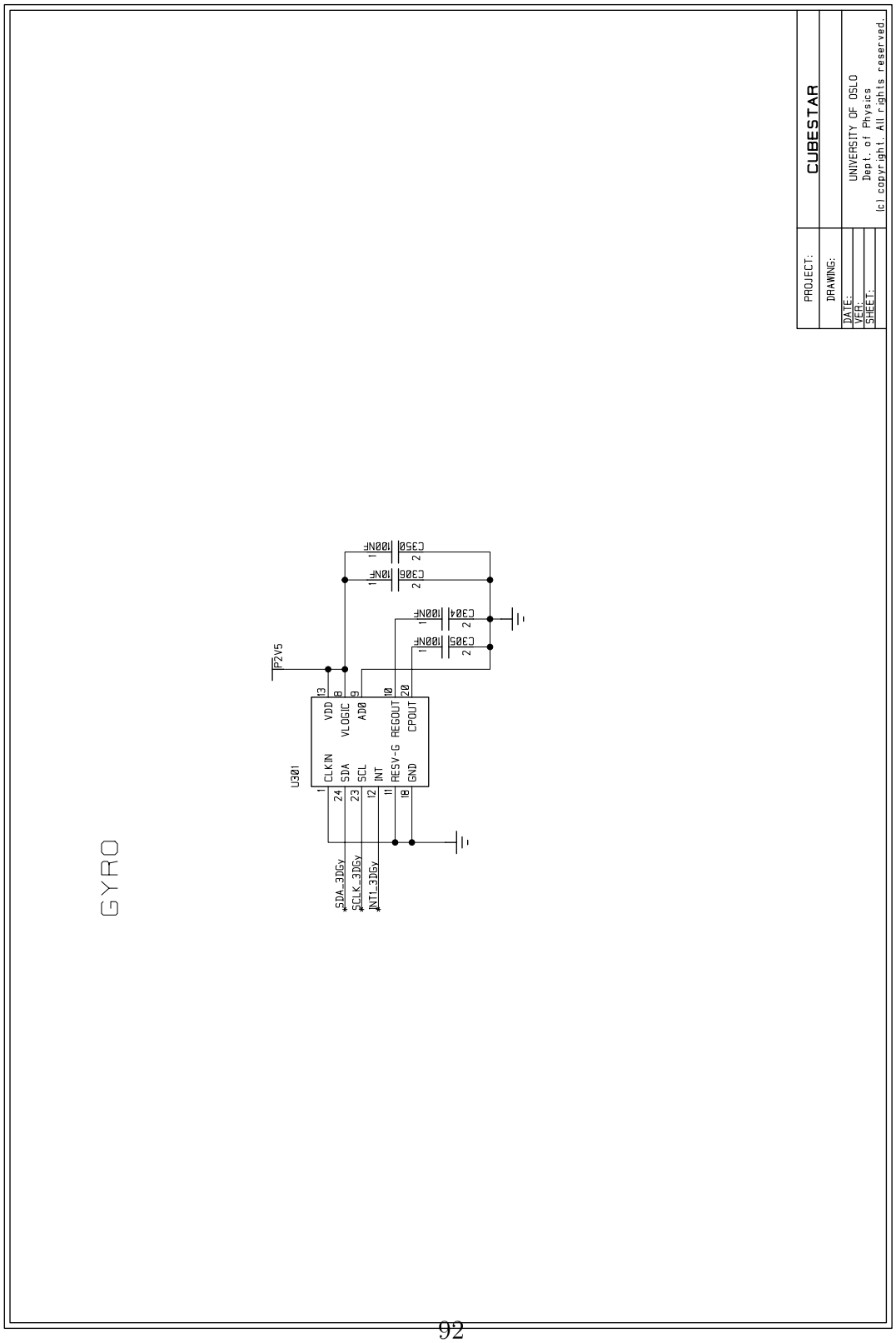
Appendix A

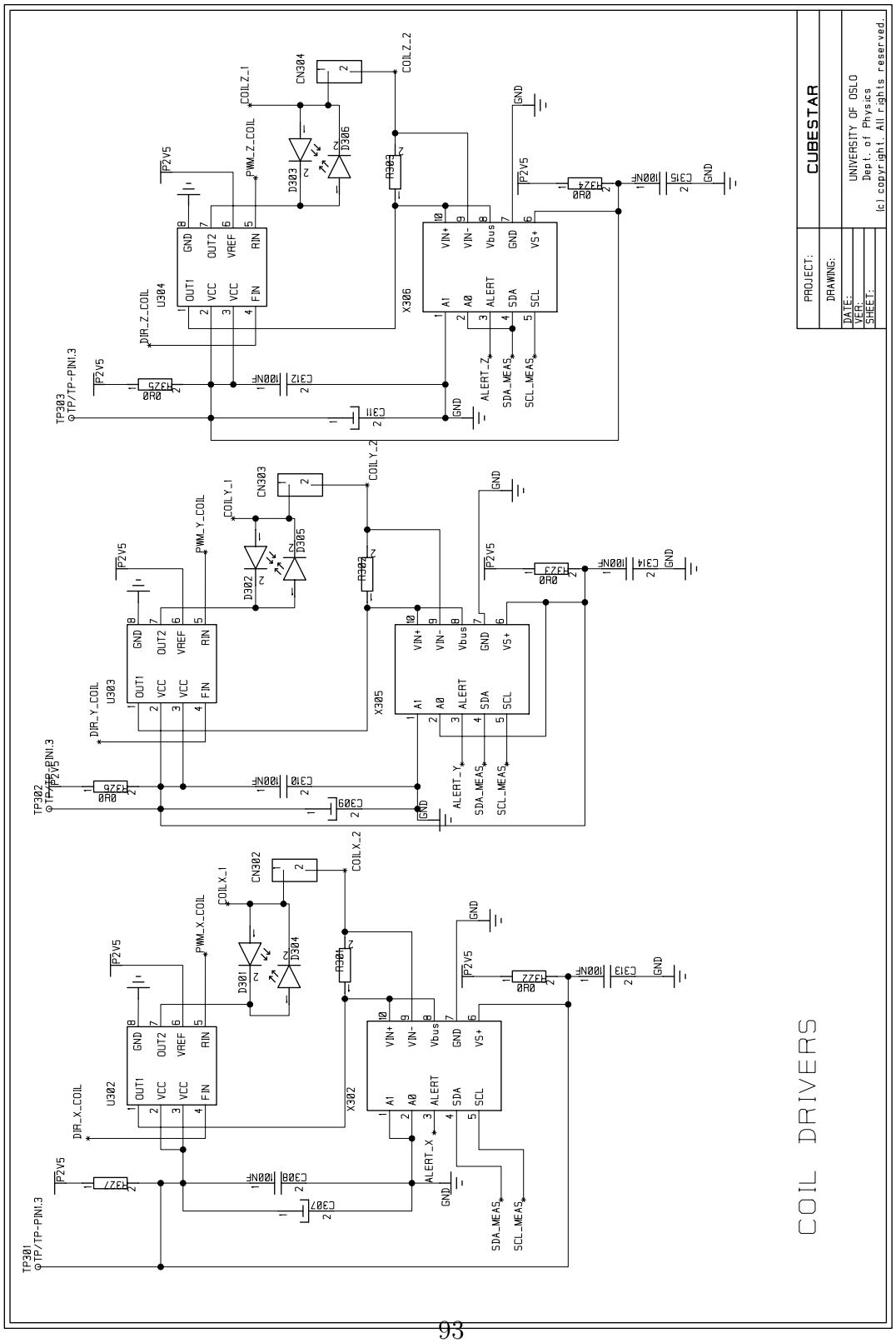
Production files

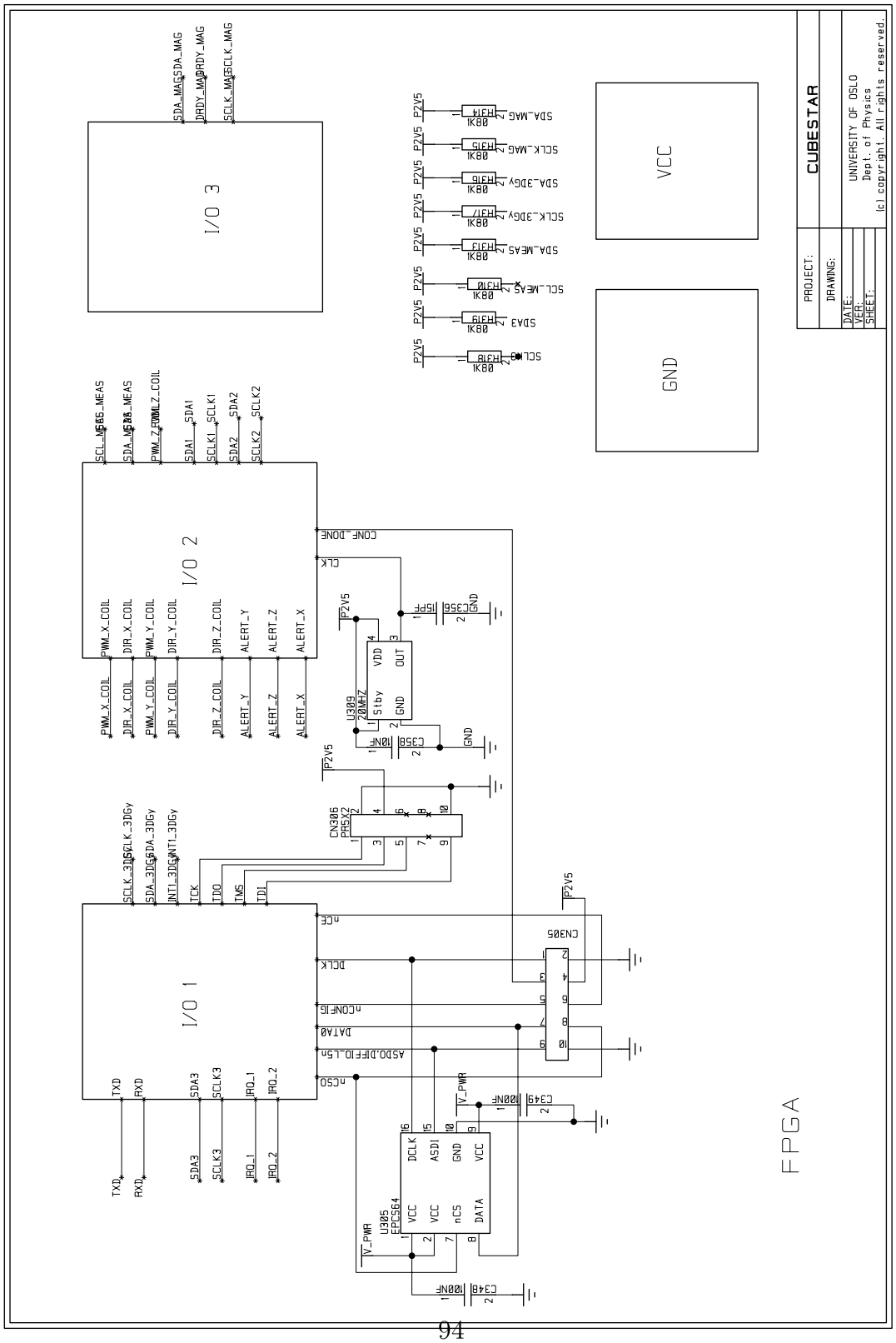
A.1 Schematics ADCS card

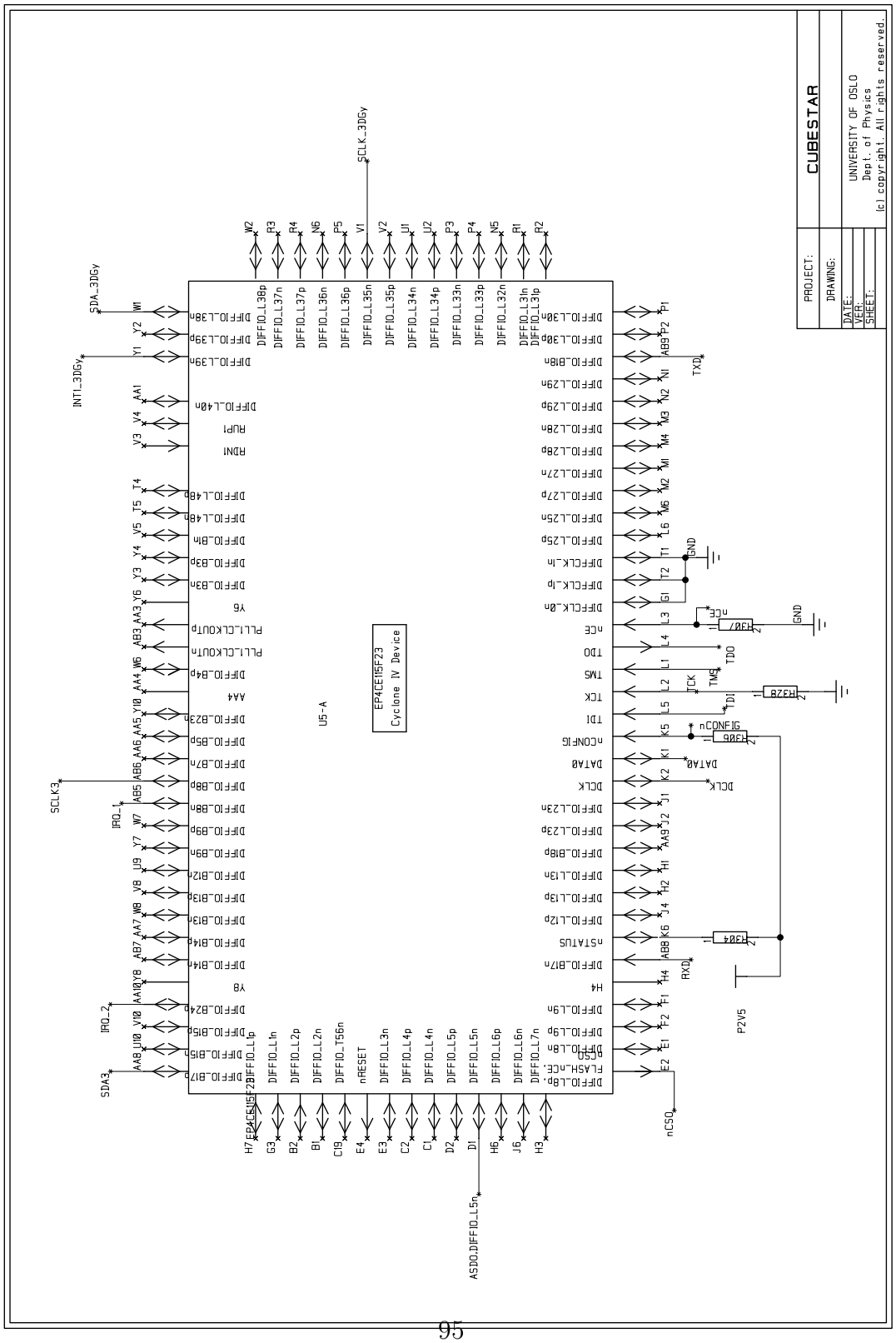

Coil Drivers
3-axis Gyro
Magneto-meter
FPGA
POWER
CN301
PROJECT: CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.

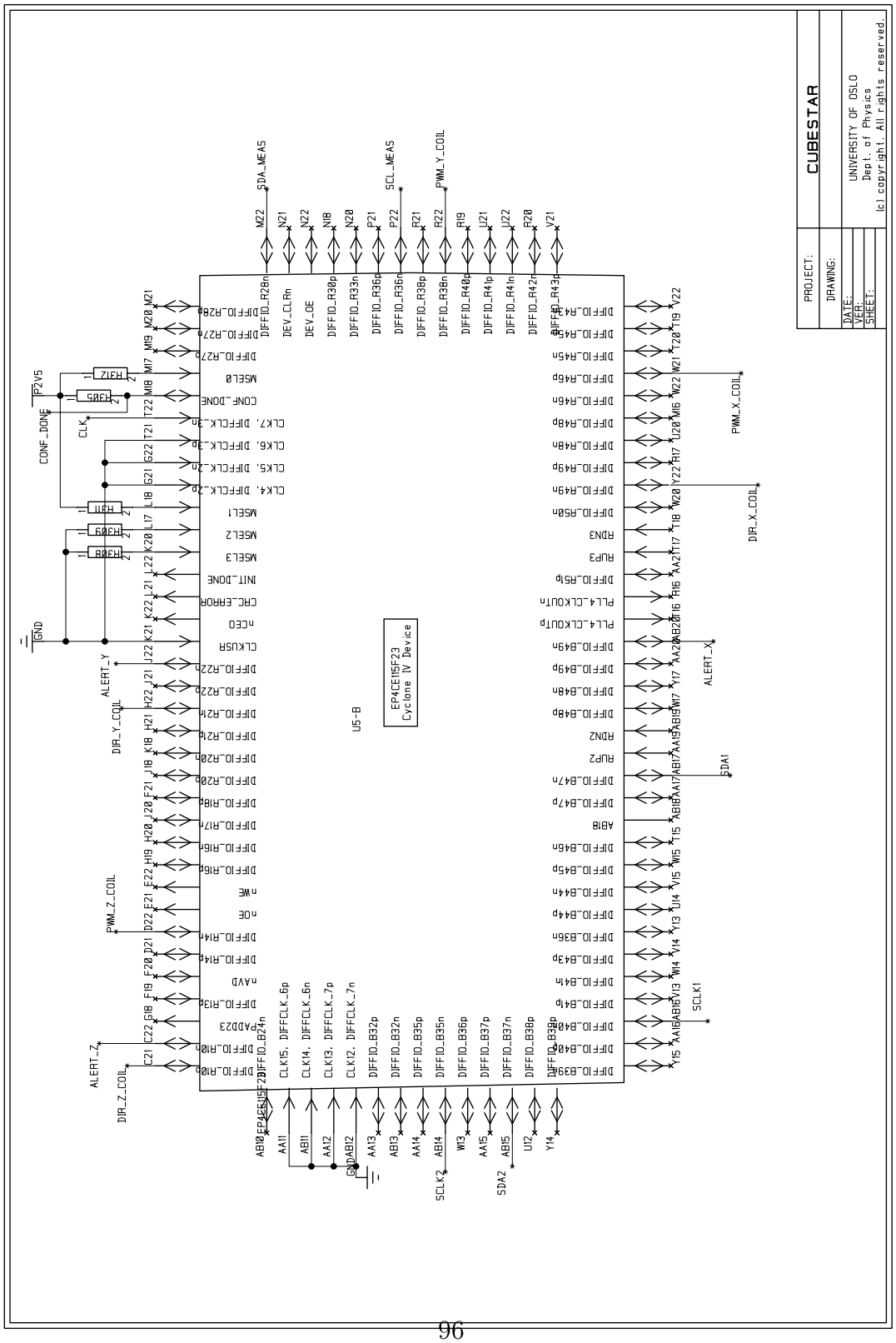

MAGNETOMETER
X301
SCL
SDA
DRDY
VDD
VDDIO
SI
SETP
SETC
GND
GND
SCLK_MAG
SDA_MAG
DRDY_MAG
P2V5
C301
C303
C302
CBI352-3531
100NF
BYPASS
PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.

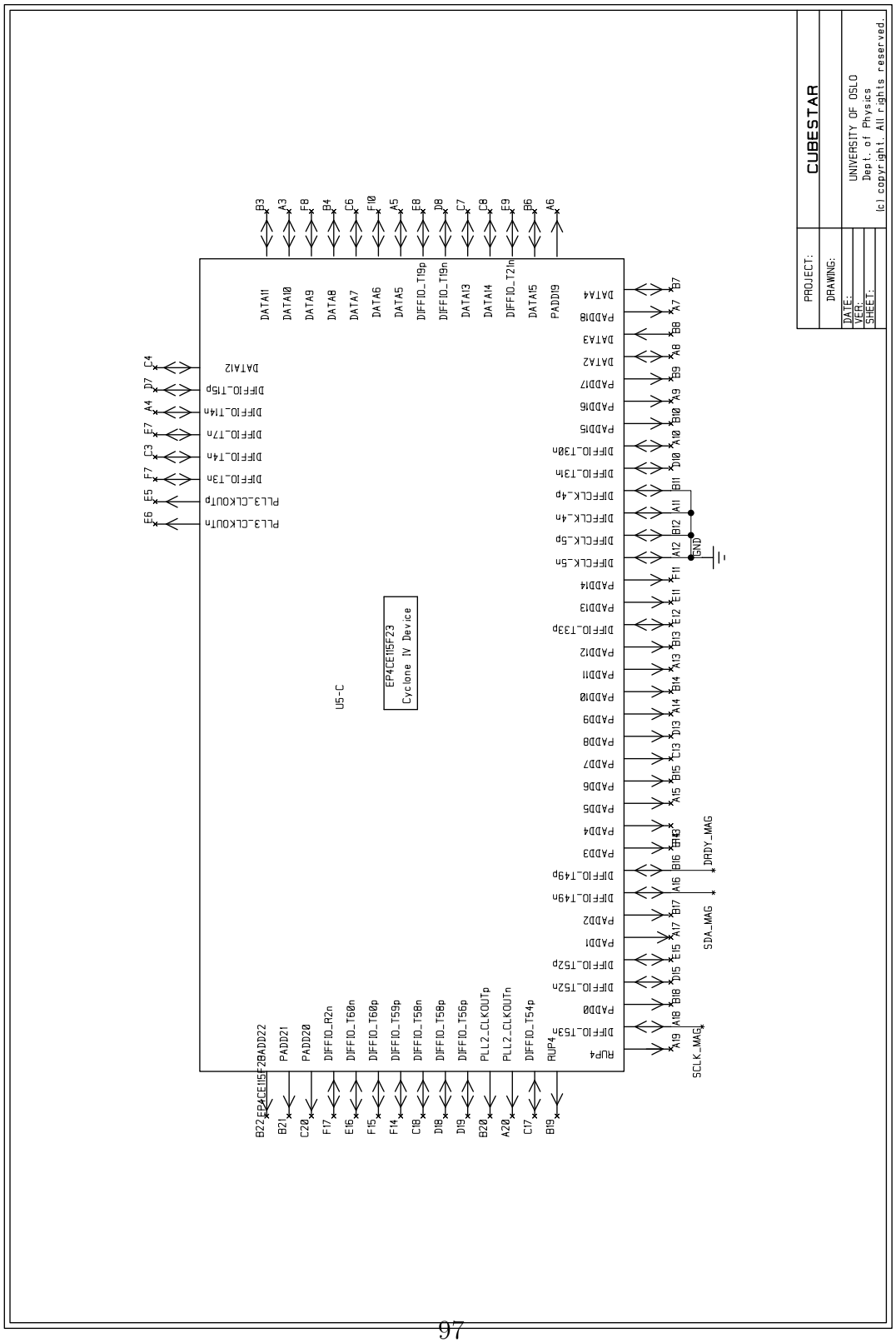

PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.
GYRO
U301
VDD
VLOGIC
AD0
REGOUT
CPOUT
CLKIN
SDA
SCL
INT
RESV-G
GND
P2V5
C350
C306
C304
C305
100NF
SDA_3DGy
SCLK_3DGy
INT1_3DGy

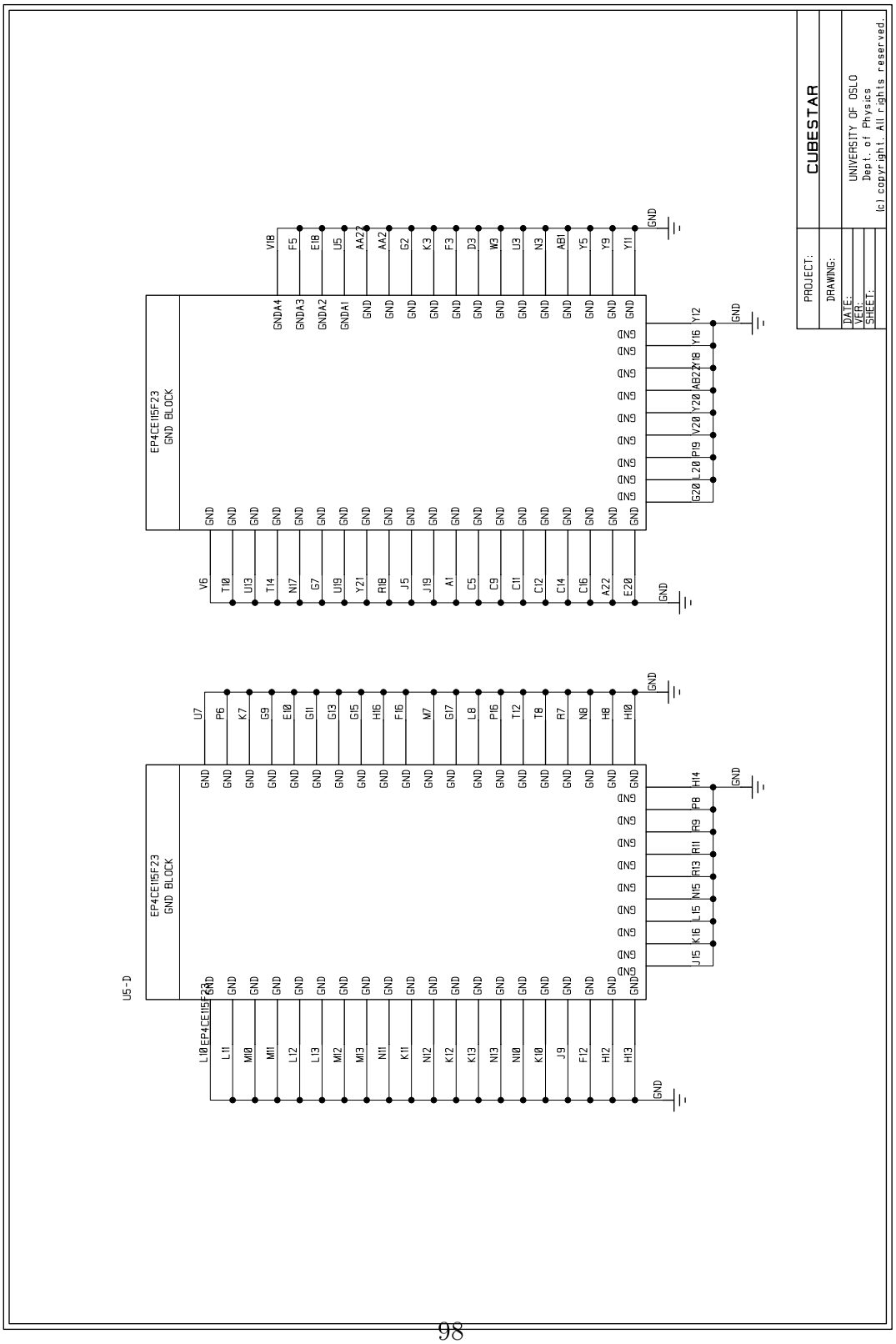

COIL DRIVERS
PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.
COILZ_2
COILZ_1
CN304
D303
D306
R303
PWM_Z_COIL
U304
DIR_Z_COIL
X306
TP303
TP/TP-PIN1.3
ALERT_Z
SDA_MEAS
SCL_MEAS
C311
C312
J24
J25
COILY_2
COILY_1
CN303
D302
D305
R302
PWM_Y_COIL
U303
DIR_Y_COIL
X305
TP302
ALERT_Y
C309
C310
J23
J26
COILX_2
COILX_1
CN302
D301
D304
R301
PWM_X_COIL
U302
DIR_X_COIL
X302
TP301
ALERT_X
C307
C308
J22
J27
P2V5
GND
0A0
100NF
GND
VIN+
VIN-
Vbus
VS+
A1
A0
ALERT
SDA
SCL
OUT1
OUT2
VREF
RIN
VCC
FIN

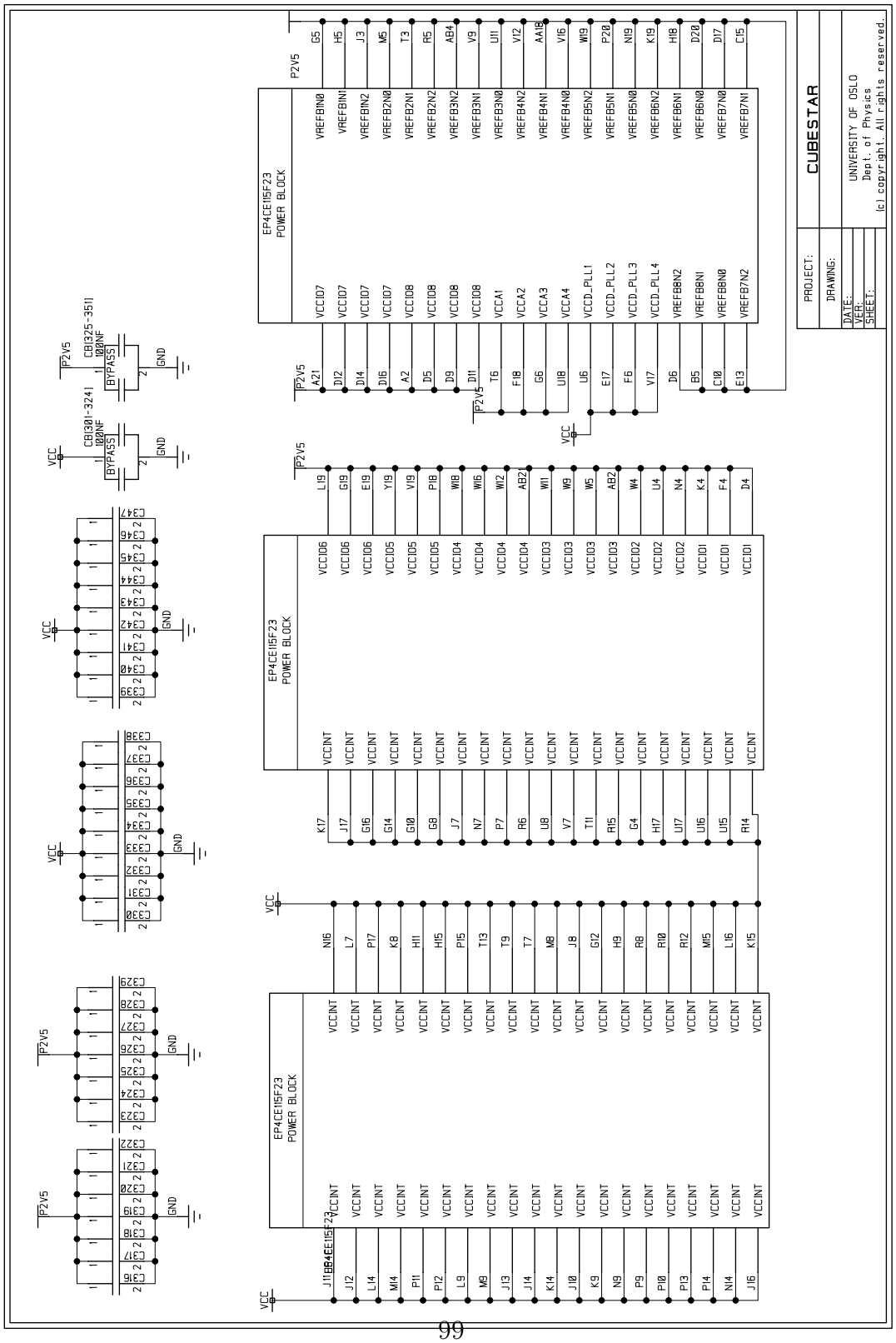

I/O 3
I/O 2
I/O 1
FPGA
VCC
GND
CUBESTAR
UNIVERSITY OF OSLO
Dept. of Physics
PROJECT:
DRAWING:
DATE:
VER:
SHEET:
SDA_MAG
DRDY_MAG
SCLK_MAG
SDA_MEAS
SCL_MEAS
PWM_Z_COIL
SDA1
SCLK1
SDA2
SCLK2
CONF_DONE
CLK
PWM_X_COIL
DIR_X_COIL
PWM_Y_COIL
DIR_Y_COIL
DIR_Z_COIL
ALERT_Y
ALERT_Z
ALERT_X
U309
20MHz
Stby
VDD
OUT
GND
C356
15PF
C358
10NF
CN306
PR5X2
SCLK_3DGy
SDA_3DGy
INT1_3DGy
TCK
TDO
TMS
TDI
nCE
DCLK
nCONFIG
DATA0
ASDO,DIFFIO_L5n
nCSO
TXD
RXD
SDA3
SCLK3
IRQ_1
IRQ_2
V_PWR
C349
100NF
U305
EPC564
DCLK
ASDI
GND
VCC
nCS
DATA
C348
CN305
P2V5
SDA_MAG
SCLK_MAG
SDA_3DGy
SCLK_3DGy
SDA_MEAS
SCL_MEAS
SDA3
SCLK3
R314
R315
R316
R317
R313
R310
R319
R318
10K
08

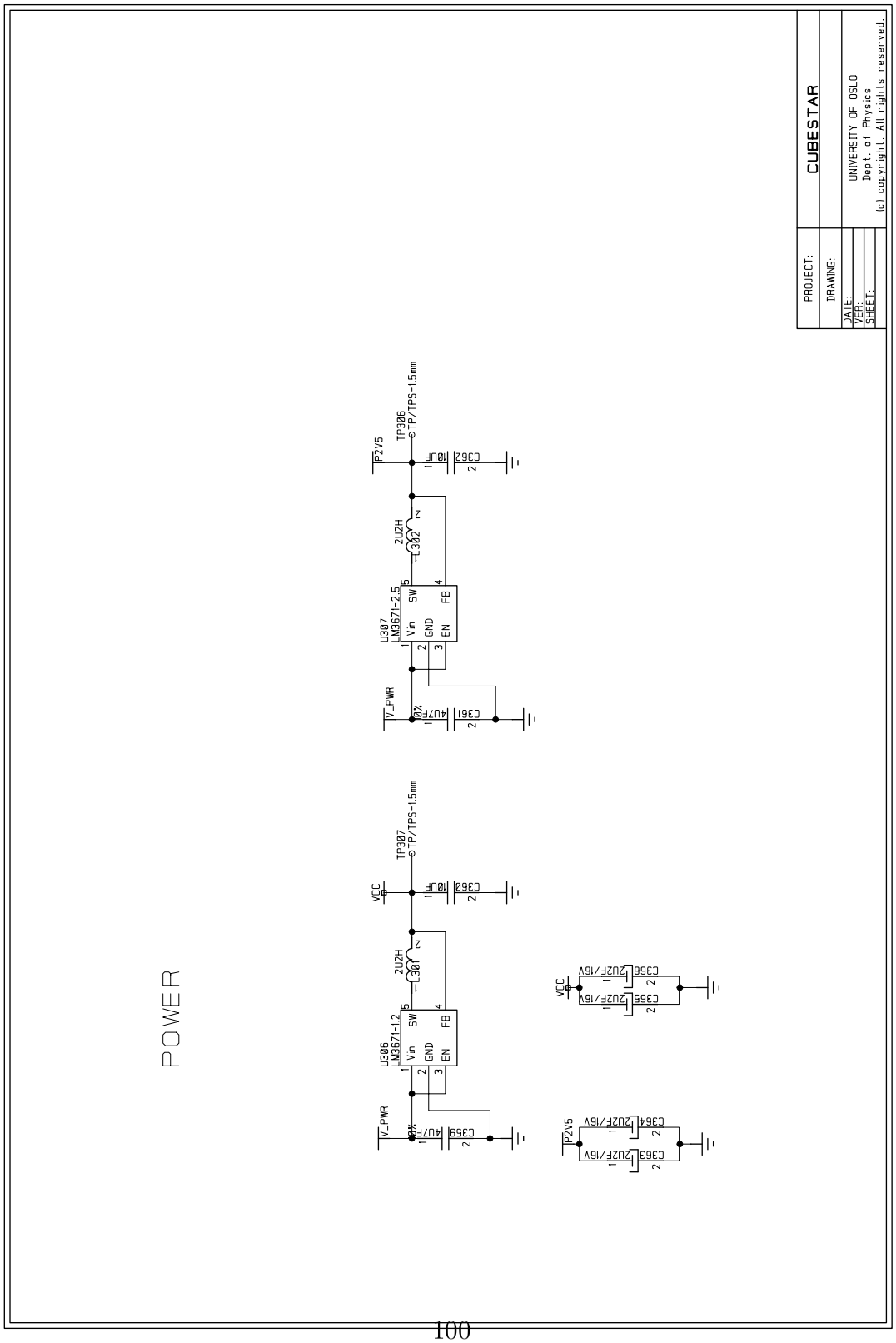

CUBESTAR
PROJECT:
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.
U5-A
EP4CE115F23
Cyclone IV Device
SCLK_3DGy
SDA_3DGy
INT1_3DGy
SCLK3
SDA3
IRQ_1
IRQ_2
TXD
RXD
TDO
TMS
TCK
TDI
nCONFIG
DATA0
DCLK
nSTATUS
nCE
nCSO
GND
P2V5
R307
R328
R306
R304
ASDO,DIFFIO_L5n


PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.
U5-B
EP4CE115F23
Cyclone IV Device
CONF_DONE
CLK
ALERT_Y
DIR_Y_COIL
PWM_Z_COIL
ALERT_Z
DIR_Z_COIL
PWM_Y_COIL
SCL_MEAS
SDA_MEAS
PWM_X_COIL
DIR_X_COIL
ALERT_X
SDA1
SCLK1
SCLK2
SDA2
GND
3V3

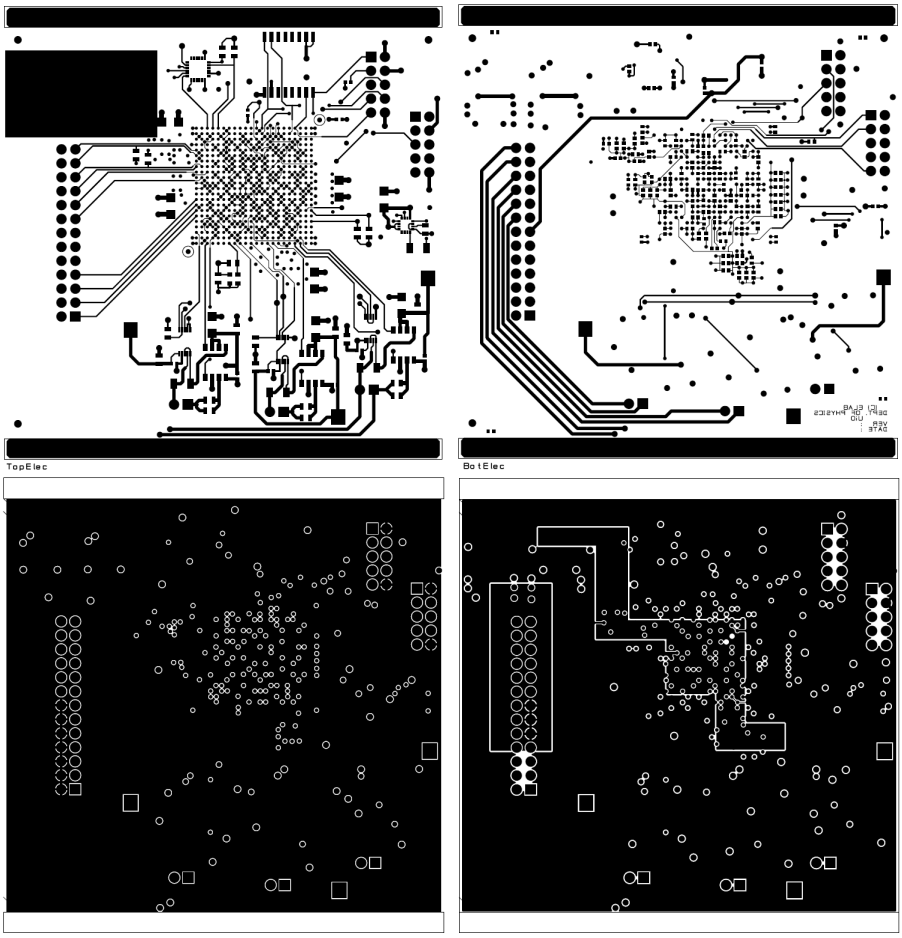

PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.
U5-C
EP4CE115F23
Cyclone IV Device
GND
DRDY_MAG
SDA_MAG
SCLK_MAG


PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.
U5-D
EP4CE115F23
GND BLOCK
GND


EP4CE115F23
POWER BLOCK
P2V5
VCC
GND
CBI325-351
CBI301-324
100NF
BYPASS
PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.


POWER
PROJECT:
CUBESTAR
DRAWING:
DATE:
VER:
SHEET:
UNIVERSITY OF OSLO
Dept. of Physics
(c) copyright. All rights reserved.
TP306
OTP/TPS-1.5mm
P2V5
10UF
C362
2U2H
L302
U307
LM3671-2.5
Vin
GND
SW
EN
FB
V_PWR
4U7F/10%
C361
TP307
OTP/TPS-1.5mm
VCC
10UF
C360
2U2H
L301
U306
LM3671-1.2
Vin
GND
SW
EN
FB
V_PWR
4U7F/10%
C359
VCC
2U2F/16V
C366
C365
P2V5
2U2F/16V
C364
C363

A.2 PCB ADCS card


TopElec
BotElec

A.3 Parts List

Design: D:\jdfk\pl5\Utlegg\ADCS_Cecilie_v16.scm

Design Title:
CUBESTAR-2012

Date: 1. februar 2013
Time: 09:39

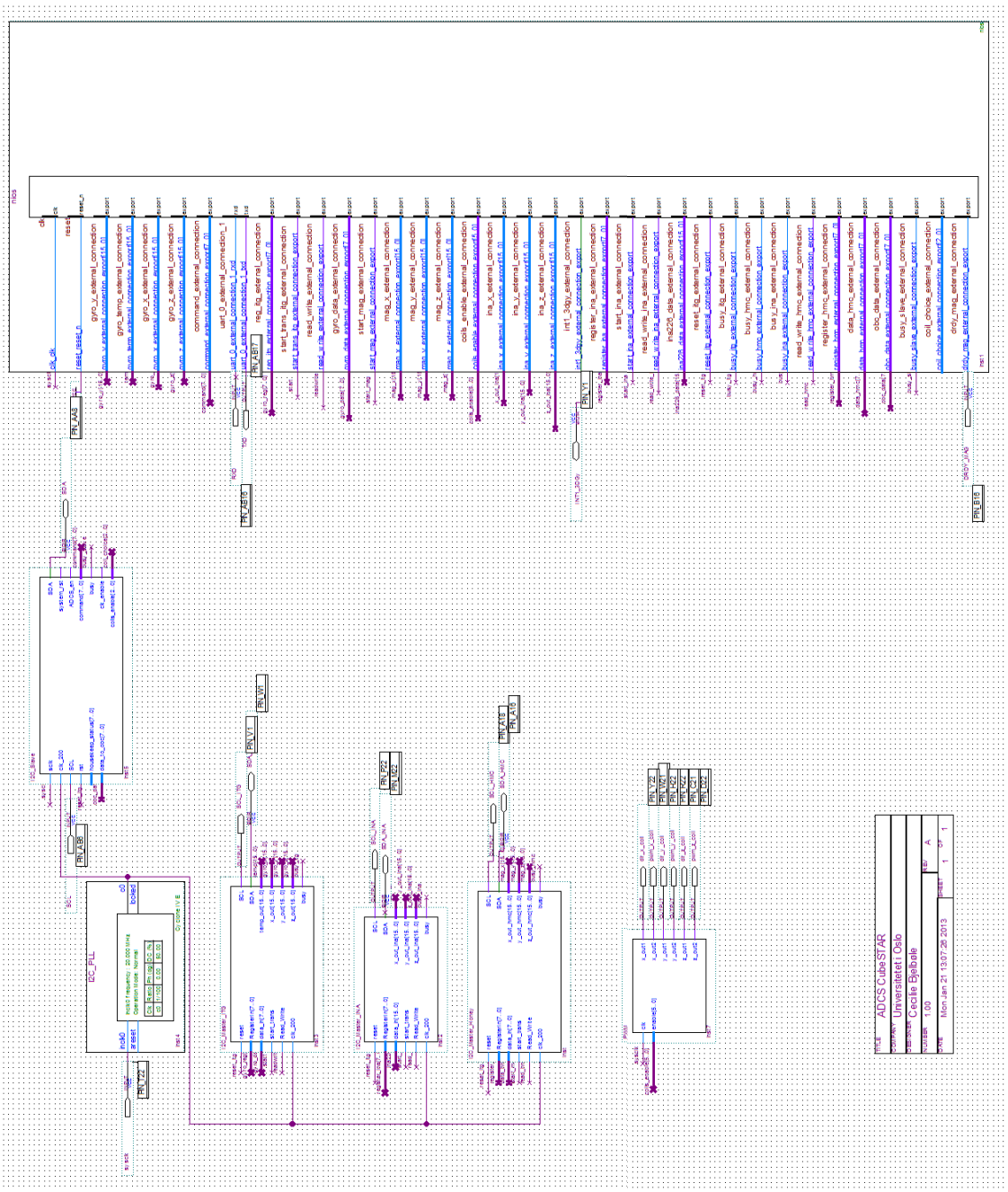
Part Name	Part Number	Description	Qty.	Comps.
ALTERA/EP4CE115F23/SMD		Cyclone IV FPGA Family	1	U5
ALTERA/EPCS64	F-1854175		1	U305
BB/INA226	F-1924807	CURRENT AND VOLTAGE MONITOR W/I2C	3	X302 X305-306
CAP/100NF/0402R	E-XX-XXX-XX	10% 16V 0402 X7R 7"REEL	27	C1-4 C304-305 C308 C310 C312-315 C322 C329 C334-338 C343-350
CAP/10NF/0402R	E-65-884-95	10% 50V 0402 X7R 7"REEL	7	C306 C320 C327 C332-333 C341-342
CAP/10NF/0603R	E-65-758-49	10% 50V 0603 X7R	1	C358
CAP/10UF/1206R	E-65-834-39	Kemet 10V 1206 Y5V	2	C360 C362
CAP/15PF/0603R	E-65-750-21	5% 50V 0603 NP0	1	C356
CAP/1N0F/0402R	E-65-883-70	10% 50V 0402 X7R 7"REEL	2	C316 C323
CAP/220NF/1206R	E-65-777-04	20% 50V 1206 X7R	1	C301
CAP/22NF/0402R	E-XX-XXX-XX	10% 50V 0402 X7R 7"REEL	6	C317-318 C321 C324-325 C328
CAP/2N2F/0402R	E-65-884-12	10% 50V 0402 X7R 7"REEL	2	C330 C339
CAP/4N7F/0402R	E-65-884-53	10% 50V 0402 X7R 7"REEL	4	C319 C326 C331 C340
CAP/4U7F/0603R	F-1833806	AVX 10% 10V 0603 X5R	3	C303 C359 C361
CAP/BYPASS/0402R	E-XX-XXX-XX	10% 16V 0603 X7R	53	CB301-353
CON/PR13X2PIN/HORIZ	E-43-714-31	13X2 TYCO PINROW ANGELED	1	CN301
CON/PR2	E-43-702-19	2 SCOTT ELEC. PINROW	3	CN302-304
CON/PR5X2	E-43-704-33	5X2 SCOTT ELEC. PINROW	2	CN305-306
IND/2U2H/1.7A	ELL4LG2R2NA	PANASONIC INDUCTOR SMD	2	L301-302
LED/19-21SDRC/SMD	E-75-308-01	SMD LED RED	3	D301-303
LED/19-21SYC/SMD	E-75-312-62	SMD LED YELLOW	3	D304-306
RES/0R00/0603R	E-60-440-02	RESISTOR KOA 0603 1% 0.1W	6	R322-327
RES/0R025/1206R	F-1703806	CURRENT SENSE RESISTOR 1206 1% 0.25W	3	R301-303
RES/10K0/0402R	E-60-889-42	RES KOA 0402 1% 63mW MINIREEL	9	R304-309 R311-312 R328
RES/1K80/0603R	E-60-448-46	RESISTOR KOA 0603 1% 0.1W	8	R310 R313-319
SPES/BD6210	F-1716258	H-BRIDGE DRIVER	3	U302-304
SPES/HMC5883L/SMD	X-XX-XXX-XX	3-AXIS DIGITAL COMPASS IC	1	X301
SPES/ITG3200/QFN	1858279	INVENSENSE 3 AXIS MEMS GYRO	1	U301
SPES/LM3671-1.2	F-1685653RL	Step-Down DC-DC Converter 1.2V	1	U306
** New part, not tested! **				
SPES/LM3671-2.5	D-LM3671MF-2.5CT-ND	Step-Down DC-DC Converter 2.5V	1	U307
** New part, not tested! **				
TANT/100UF/6V3/SMD	F-1135257	AVX TANTAL ELECTROLYTIC CAP	4	C302 C307 C309 C311
TANT/2U2F/16V/SMD	E-67-737-82	TANTAL ELECTROLYTIC CAP	4	C363-366
XTAL/ASEMB20MHZ	D-535-11122-6-ND	ABRACON MEMS CLOCK OSCILLATOR	1	U309
** Not Verified **				

End of report

Appendix B

VHDL code

B.1 Top File


ADCS CubeSTAR
Universitetet i Oslo
Cecilie Bjelobe
Mon Jan 21 13:07:28 2019

B.2 Magnetometer I2C driver

```
-- Author      : Cecilie Bjelbole
-- Company     : University of Oslo
-- File name   : I2C_Master_HMC.vhd
-- Date        : 28.08.2012
-- Project     : ADCS for CubeSTAR
-- Function    : I2C MASTER DRIVER FOR THE HMC MAGNETOMETER

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity I2C_Master_Honey is
        port(
        SCL                     : out std_logic;
        SDA                     : inout std_logic;

        reset                   : in std_logic;
        RegisterIn              : in std_logic_vector(7 downto 0);
        data_in                 : in std_logic_vector(7 downto 0);

        x_out_hmc               : out std_logic_vector(15 downto 0);
        y_out_hmc               : out std_logic_vector(15 downto 0);
        z_out_hmc               : out std_logic_vector(15 downto 0);
        start_trans             : in std_logic;
        Read_Write              : in std_logic; ---Read_Write = 1 when read
        clk_200                 : in std_logic;

        busy                    : out std_logic

        );
        end I2C_Master_Honey;


architecture I2C_Master_Honey_arch of I2C_Master_Honey is
type statetype is (idle, waiting, start, write_address, write_address2,
    ACK_init, ACK, send_reg_address, send_reg_address2, ACK_init2, ACK2,
    do_write, do_write2, ACK_init3, ACK3,
send_stop_init, send_stop, stop, do_read_init, start_1, start_2,
    new_address, new_address2, ACK_init4,
```

```
ACK4, receive_data, receive_data1, receive_data2, received_bytes1,
    received_bytes, nack, stop_init, stop1, stop2, error);

-------------------------SIGANL
    DECLARATIONS-----------------------------------

signal SDA_1                          : std_logic;
signal clk_100                        : std_logic;
signal shift_en                       : std_logic;  --Trigger for the
    shift register
signal bit_counter                    : integer range 0 to 7:=7;
signal byte_counter                   : integer range 0 to 47:=0;
signal state                          : statetype;
signal SensorData                     : std_logic_vector(15 downto 0);
signal SensorData_in                  : std_logic_vector(15 downto 0);

signal dat_in                         : std_logic_vector(7 downto 0);



------------------------------ CONSTANTS
    -------------------------------------
constant SlaveAddress_Read:  std_logic_vector(7 DOWNTO 0) :=
    "00111101"; ---0x3D -read adress honeywell "00111101"
constant SlaveAddress_Write:  std_logic_vector(7 DOWNTO 0) :=
    "00111100"; ---0x3C -write adress honeywell "00111100"




 -------------------------------- Components
     ---------------------------------
 ------------------------------- Shift Register
     --------------------------------
component SR_SerIn_redge is
  generic (
    width : integer := 16);
  port
    (
    clk     : in std_logic;
    DataIn  : in std_logic;
    shift_en : in std_logic;
    DataOut : out std_logic_vector(width-1 downto 0)
```

```
    );
end component SR_SerIn_redge;


begin
----- Port Mapping ------

SR: entity work.SR_SerIn_redge(SR_SerIn_arch)
        port map(       clk => clk_100,
                                DataIn => SDA,
                                shift_en => shift_en,
                                DataOut => SensorData_in
                        );
                        SensorData <= SensorData_in(15 downto 0);




----- Set high to Z ------
SDA <= 'Z' when SDA_1 = '1' else '0';
SCL <= clk_100;

statm: process(clk_200)
begin
        if rising_edge(clk_200) then
        busy <= '1';
                if reset = '1' then
                        clk_100 <= '1'; ---Set SCL to clk_100
                        SDA_1 <= '1';
                        bit_counter <= 7; ---Start the bit counter at 7
                        byte_counter <= 0;
                        shift_en <= '0';
                        x_out_hmc <= "0000000000000000";
                        y_out_hmc <= "0000000000000000";
                        z_out_hmc <= "0000000000000000";
                        state <= idle;
                else
                        shift_en <= '0';
                case state is
                --------- IDLE MODE -------------
                        when idle =>
                        clk_100 <= '1';
```

```
        SDA_1 <= '1';
        busy <= '0';
        state <= waiting;

--------- WAITING ---------------
        when waiting =>
        clk_100 <= '1';
        SDA_1 <= '1';
        busy <= '0';
        if start_trans = '1' then
                state <= start;
        else
                state <= waiting;
        end if;

--------- SENDING START CONDITION ----------
        when start => --- send start condition: Pull SCL
            high while SDA goes low
        clk_100 <= '1';
        SDA_1 <= '0';
        state <= write_address;

--------- WRITE SLAVE ADDRESS + write condition
    -----------
        when write_address => -- send 7 bit adress + R/W
            MSB sends first. SDA can only change when when
            SCL is at low
        clk_100 <= '0';
        SDA_1 <= SlaveAddress_Write(bit_counter); --Send
            the slave address
        state <= write_address2;

        when write_address2 =>
        clk_100 <= '1';
        if bit_counter > 0 then
                bit_counter <= bit_counter - 1;
                state <= write_address;
        else
                bit_counter <= 7;
                state <= ACK_init;
        end if;

-------- GET ACKNOWLEDGE FROM SLAVE ------------
```

```
            when ACK_init => --- Set SCL low and SDA high
                before waiting for the ACK bit from the slave
            clk_100 <= '0';
            SDA_1 <= '1';
            state <= ACK;

            when ACK =>     --- Receiving ACK from slave
            clk_100 <= '1';
            if SDA <= '0' then
                    state <= send_reg_address;
            else ------ ACK-error if SDA=1
                    state <= error;
            end if;

        --------- SEND REGISTER ADDRESS TO SLAVE ----------
            when send_reg_address =>
            clk_100 <= '0';
            SDA_1 <= RegisterIn(bit_counter); ----Register
                adress sent in with RegisterIn
            state <= send_reg_address2;

            when send_reg_address2 => ---continue register
                adress sending
            clk_100 <= '1';
                    if bit_counter > 0 then
                            bit_counter <= bit_counter -1;
                            state <= send_reg_address;
                    else
                            bit_counter <= 7;
                            state <= ACK_init2;
                    end if;

            ------ get ack
            when ACK_init2 => --- Set SCL low and SDA high
                before waiting for the ACK bit from the slave
            clk_100 <= '0';
            SDA_1 <= '1';
            state <= ACK2;

            when ACK2 =>   --- Receiving ACK from slave
            clk_100 <= '1';
            if SDA = '0' then
```

```
                    if Read_Write = '0' then ----Read_Write =
                        1 when read
                    state <= do_write;
                    else
                    state <= do_read_init; --- send repeated
                        start and then go do a read
                    end if;
                else state <= error; ----- ACK-error if SDA=1
                end if ;

        --------- WRITE TO THE SLAVE -------------
                when do_write =>
                clk_100 <= '0';
                SDA_1 <= data_in(bit_counter);
                state <= do_write2;

                when do_write2 =>
                clk_100 <= '1';
                if bit_counter > 0 then
                        bit_counter <= bit_counter -1;
                        state <= do_write;
                else
                        bit_counter <= 7;
                        state <= ACK_init3;
                end if;

                ------ get ack
                --- Set SCL low and SDA high before waiting for
                    the ACK bit from the slave
                when ACK_init3=>
                clk_100 <= '0';
                SDA_1 <= '1';
                state <= ACK3;

                when ACK3 =>    --- Receiving ACK from slave
                clk_100 <= '1';
                if SDA = '0' then
                        state <= send_stop_init;
                else
                        state <= error;
                end if;
```

```
            ------ send stop
            when send_stop_init => ---- Prepare the stop
                condition. SDA and SCL is 0
            clk_100 <= '0';
            SDA_1 <= '0';
            state <= send_stop;

            when send_stop =>
            clk_100 <= '1';      ------ SCL is high for the
                stop
            SDA_1 <= '0';        ------ while the SDA goes
                from 0...
            state <= stop;

            when stop =>
            clk_100 <= '1';  ------ SCL remains high,
            SDA_1 <= '1';    -- while SDA changes to high.
            state <= idle;   -- both lines are at high = we
                are in idle.



        --------- READ FROM THE SLAVE -------------
        ----- need to send a repeated start signal after the
            address write.
            when do_read_init => -- SDA starts at 1 to prepare
                for the 1 to 0 transition
            clk_100 <= '0';
            SDA_1 <= '1';
            state <= start_1;

            when start_1 =>
            clk_100 <= '1';
            SDA_1 <= '1';
            state <= start_2;

            when start_2 =>
            clk_100 <= '1';  ----- SCL stays at 1 while
            SDA_1 <= '0';    ----- SDA changes from high to low
            bit_counter <= 7;
            state <= new_address;

        ----- sending the slave address
```

```
            when new_address =>
            clk_100 <= '0';
            SDA_1 <= SlaveAddress_Read(bit_counter);
            state <= new_address2;

            when new_address2 =>
            clk_100 <= '1';
            if bit_counter > 0 then
                    bit_counter <= bit_counter - 1;
                    state <= new_address;
            else
                    bit_counter <= 7;
                    state <= ACK_init4;
            end if;

        ------ get ack
            when ACK_init4 =>
            clk_100 <= '0';
            SDA_1 <= '1';
            state <= ACK4;

            when ACK4 =>
            clk_100 <= '1';
            if      SDA = '0' then
                    bit_counter <= 7;
                    state <= receive_data; ---- go to receive
                        slave
            else
                    state <= error; ----- Ack-error
            end if;

        --------- RECEIVE FROM THE SLAVE -------------
            when receive_data =>
            clk_100 <= '1';
            state <= receive_data1;

            when receive_data1 =>
            clk_100 <= '0';
            SDA_1 <= '1';
            state <= receive_data2;

            when receive_data2 =>
            clk_100 <= '1';
```

```
shift_en <= '1';
---- First two bytes x register ----
if byte_counter = 7 then
       state <= received_bytes1;
elsif byte_counter = 15 then
       state <= received_bytes;
---- 3 and 4 is magnetometer y register ----
elsif byte_counter = 23 then
       state <= received_bytes1;
elsif byte_counter = 31 then
       state <= received_bytes;
---- 5 and 6 is magnetometer z register ----
elsif byte_counter = 39 then
       state <= received_bytes1;
elsif byte_counter = 47 then
       state <= received_bytes;
----- else: clock in more data -----
else
       byte_counter <= byte_counter + 1;
       state <= receive_data1;
end if;

----When received one byte of data, send ack bit.
when received_bytes1 =>
clk_100 <= '0';
SDA_1 <= '0';
byte_counter <= byte_counter + 1;
state <= receive_data;

when received_bytes =>
clk_100 <= '0';
SDA_1 <= '0';
if byte_counter = 15 then
       x_out_hmc <= SensorData;
       byte_counter <= byte_counter + 1;
       state <= receive_data;
elsif byte_counter = 31 then
       y_out_hmc <= SensorData;
       byte_counter <= byte_counter + 1;
       state <= receive_data;
elsif byte_counter = 47 then
       z_out_hmc <= SensorData;
       byte_counter <= 0;
```

```
                        state <= nack;
                    else
                        state <= error;
                    end if;

                    when nack =>    -----Send Not Aknowledge
                        bit,NACK(SCL is high)
                    clk_100 <= '1';
                    state <= stop_init;

                --------- SEND STOP CONDITION -------------
                    ---- SDA goes from low to high, while SCL is high.
                    when stop_init =>  ----prepare for the stop. set
                        to low
                    clk_100 <= '0';
                    SDA_1 <= '0';
                    state <= stop1;

                    when stop1 =>
                    clk_100 <= '1';
                    SDA_1 <= '0'; ----SDA goes from 0 to 1, while SCL
                        is 1-
                    state <= stop2;

                    when stop2 =>
                    clk_100 <= '1';
                    SDA_1 <= '1';
                    state <= idle;

                    when error =>
                    state <= waiting;

                end case;
                end if;
        end if;

end process;

end I2C_Master_Honey_arch;
```

B.3 Current Sensing I2C driver

```
-- Author      : Cecilie Bjelbole
-- Company     : University of Oslo
-- File name   : I2C_Master_INA.vhd
-- Date        : 11.09.2012
-- Project     : ADCS for CubeSTAR
-- Function    : I2C MASTER DRIVER FOR CURRENT SENSOR

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity I2C_Master_INA is
        port(
        SCL                             : out std_logic;
        SDA                             : inout std_logic;

        reset                   : in std_logic;
        RegisterIn              : in std_logic_vector(7 downto 0);
        data_in                 : in std_logic_vector(15 downto 0);

        x_out_ina               : out std_logic_vector(15 downto 0);
        y_out_ina               : out std_logic_vector(15 downto 0);
        z_out_ina               : out std_logic_vector(15 downto 0);
        start_trans             : in std_logic;
        Read_Write              : in std_logic; ---Read_Write = 1 when read
        clk_200                 : in std_logic;
        busy                    : out std_logic


        );
        end I2C_Master_INA;


architecture I2C_Master_INA_arch of I2C_Master_INA is
type statetype is (idle, waiting, start, write_address, write_address2,
    ACK_init, ACK, send_reg_address, send_reg_address2, ACK_init2, ACK2,
    do_write, do_write2, ACK_init3, ACK3,
ACK32, ACK_init32, do_write2_part2, do_write_part2,
```

```
send_stop_init, send_stop, stop, do_read_init, start_1, start_2,
    new_address, new_address2, ACK_init4,
ACK4, receive_data, receive_data1, receive_data2, received_bytes1,
    received_bytes, nack, stop_init, stop1, stop2,error);

-------------------------SIGANL
    DECLARATIONS-----------------------------------

signal SDA_1                                : std_logic;
signal clk_100                              : std_logic;
signal shift_en                             : std_logic;  --Trigger for
    the shift register
signal bit_counter                          : integer range 0 to 7:=7;
signal byte_counter                 : integer range 0 to 15:=0;
signal state                                : statetype;
signal SensorData                           : std_logic_vector(15 downto
    0);
signal SensorData_in                : std_logic_vector(15 downto 0);
signal data_ready                           : integer range 0 to 2:=0;
    --counter to control slave device

signal data_in_high    : std_logic_vector(7 downto 0);
signal data_in_low     : std_logic_vector(7 downto 0);

------------------------------ CONSTANTS
    -------------------------------------
constant SlaveAddress_Read_X:  std_logic_vector(7 DOWNTO 0) :=
    "10000001"; --- read adress INA226_x "10000001"
constant SlaveAddress_Write_X:  std_logic_vector(7 DOWNTO 0) :=
    "10000000"; --- write adress INA226_x "10000000"
constant SlaveAddress_Read_Y:  std_logic_vector(7 DOWNTO 0) :=
    "10000011";
constant SlaveAddress_Write_Y:  std_logic_vector(7 DOWNTO 0) :=
    "10000010";
constant SlaveAddress_Read_Z:  std_logic_vector(7 DOWNTO 0) :=
    "10000101";
constant SlaveAddress_Write_Z:  std_logic_vector(7 DOWNTO 0) :=
    "10000100";


 -------------------------------- Components
     ---------------------------------
```

```
 ------------------------------ Shift Register
    ------------------------------
component SR_SerIn_redge is
  generic (
    width : integer := 16);
  port
    (
    clk     : in  std_logic;
    DataIn  : in  std_logic;
    shift_en : in std_logic;
    DataOut : out std_logic_vector(width-1 downto 0)
    );
end component SR_SerIn_redge;


begin
----- Port Mapping ------

SR: entity work.SR_SerIn_redge(SR_SerIn_arch)
       port map(      clk => clk_100,
                                DataIn => SDA,
                                shift_en => shift_en,
                                DataOut => SensorData_in
                      );
                      SensorData <= SensorData_in(15 downto 0);


----- Set high to Z ------
SDA <= 'Z' when SDA_1 = '1' else '0';
SCL <= clk_100;
data_in_high <= data_in(15 downto 8);
data_in_low <= data_in(7 downto 0);
statm: process(clk_200)
begin
       if rising_edge(clk_200) then
       busy <= '1';
              if reset = '1' then
                     clk_100 <= '1'; ---Set SCL to clk_100
                     SDA_1 <= '1';
                     bit_counter <= 7; ---Start the bit counter at 7
                     byte_counter <= 0;
                     shift_en <= '0';
                     x_out_ina <= "0000000000000000";
```

```
        y_out_ina <= "0000000000000000";
        z_out_ina <= "0000000000000000";
        data_ready <= 0;
        state <= idle;
else
        shift_en <= '0';
case state is
--------- IDLE MODE -------------
        when idle =>
        busy <= '0';
        clk_100 <= '1';
        SDA_1 <= '1';
        state <= waiting;

--------- WAITING ---------------
        when waiting =>
        clk_100 <= '1';
        SDA_1 <= '1';
        if start_trans = '1' then
                state <= start;
        else
                state <= waiting;
        end if;

--------- SENDING START CONDITION ----------
        when start =>
 --- send start condition:
 --Pull SCL high while SDA goes low
        clk_100 <= '1';
        SDA_1 <= '0';
        state <= write_address;

--------- WRITE SLAVE ADDRESS + write condition
    -----------
        when write_address => -- send 7 bit adress + R/W
            MSB sends first.
        --SDA can only change when when SCL is at low
        clk_100 <= '0';
        if data_ready = 0 then
                SDA_1 <=
                    SlaveAddress_Write_X(bit_counter);
                    --Send the slave address X axis
                state <= write_address2;
```

```
        elsif data_ready = 1 then
                SDA_1 <= SlaveAddress_Write_Y(bit_counter);
                state <= write_address2;
        elsif data_ready = 2 then
                SDA_1 <= SlaveAddress_Write_Z(bit_counter);
                state <= write_address2;
        end if;

        when write_address2 =>
        clk_100 <= '1';
        if bit_counter > 0 then
                bit_counter <= bit_counter - 1;
                state <= write_address;
        else
                bit_counter <= 7;
                state <= ACK_init;
        end if;

-------- GET ACKNOWLEDGE FROM SLAVE ------------
        when ACK_init =>
--- Set SCL low and SDA high before waiting for the ACK
    bit from the slave
        clk_100 <= '0';
        SDA_1 <= '1';
        state <= ACK;

        when ACK =>     --- Receiving ACK from slave
        clk_100 <= '1';
        if SDA <= '0' then
                state <= send_reg_address;
        else ------ ACK-error if SDA=1
                state <= error;
        end if;

--------- SEND REGISTER ADDRESS TO SLAVE ----------
        when send_reg_address =>
        clk_100 <= '0';
        SDA_1 <=       RegisterIn(bit_counter);
            ----Register adress sent in with RegisterIn
        state <= send_reg_address2;

        when send_reg_address2 => ---continue register
            adress sending
```

```
clk_100 <= '1';
        if bit_counter > 0 then
                bit_counter <= bit_counter -1;
                state <= send_reg_address;
        else
                bit_counter <= 7;
                state <= ACK_init2;
        end if;

------ get ack
 --- Set SCL low and SDA high before waiting for
     the ACK bit from the slave
when ACK_init2 =>
clk_100 <= '0';
SDA_1 <= '1';
state <= ACK2;

when ACK2 =>   --- Receiving ACK from slave
clk_100 <= '1';
if SDA = '0' then
        if Read_Write = '0' then ----Read_Write =
            1 when read
        state <= do_write;
        else
        state <= do_read_init; --- send repeated
            start and then go do a read
        end if;
else state <= error; ----- ACK-error if SDA=1
end if ;

--------- WRITE TO THE SLAVE -------------
        when do_write =>
        clk_100 <= '0';
        SDA_1 <= data_in_high(bit_counter);
        state <= do_write2;


        when do_write2 =>
        clk_100 <= '1';
        if bit_counter > 0 then
                bit_counter <= bit_counter -1;
                state <= do_write;
        else
```

```
        bit_counter <= 7;
        state <= ACK_init3;
end if;

------ get ack
--- Set SCL low and SDA high before waiting for
    the ACK bit from the slave
when ACK_init3=>
clk_100 <= '0';
SDA_1 <= '1';
state <= ACK3;

when ACK3 =>     --- Receiving ACK from slave
clk_100 <= '1';
if SDA = '0' then
        state <= do_write_part2;--send_stop_init;
else
        state <= error;
end if;

when do_write_part2 =>
clk_100 <= '0';
SDA_1 <= data_in_low(bit_counter);
state <= do_write2_part2;

when do_write2_part2 =>
clk_100 <= '1';
if bit_counter > 0 then
        bit_counter <= bit_counter -1;
        state <= do_write_part2;
else
        bit_counter <= 7;
        state <= ACK_init32;
end if;
------ get ack
when ACK_init32=>
clk_100 <= '0';
SDA_1 <= '1';
state <= ACK32;

when ACK32 =>     --- Receiving ACK from slave
clk_100 <= '1';
if SDA = '0' then
```

```
                state <= send_stop_init;
            else
                state <= error;
            end if;

            ------ send stop
            when send_stop_init => ---- Prepare the stop
                condition. SDA and SCL is 0
            clk_100 <= '0';
            SDA_1 <= '0';
            state <= send_stop;

            when send_stop =>
            clk_100 <= '1';      ------ SCL is high for the
                stop
            SDA_1 <= '0';        ------ while the SDA goes
                from 0...
            state <= stop;

            when stop =>
            clk_100 <= '1';          ------ SCL remains high,
            SDA_1 <= '1';        ------ while SDA changes to
                high.
            state <= idle;       ------ both lines are at high
                = we are in idle.



        --------- READ FROM THE SLAVE -------------
        ----- send a repeated start signal after the address
            write.
                when do_read_init => ----- SDA starts at 1 to
                    prepare for the 1 to 0 transition
                clk_100 <= '0';
                SDA_1 <= '1';
                state <= start_1;

                when start_1 =>
                clk_100 <= '1';
                SDA_1 <= '1';
                state <= start_2;

                when start_2 =>
```

```
                clk_100 <= '1';  ----- SCL stays at 1 while
                SDA_1 <= '0';    ----- SDA changes from high to low
                bit_counter <= 7;
                state <= new_address;

        ----- sending the slave address
                when new_address =>
                clk_100 <= '0';
                if data_ready = 0 then
                        SDA_1 <= SlaveAddress_Read_X(bit_counter);
                            --Send the slave address X axis
                elsif data_ready = 1 then
                        SDA_1 <= SlaveAddress_Read_Y(bit_counter);
                elsif data_ready = 2 then
                        SDA_1 <= SlaveAddress_Read_Z(bit_counter);
                end if;
                state <= new_address2;

                when new_address2 =>
                clk_100 <= '1';
                if bit_counter > 0 then
                        bit_counter <= bit_counter - 1;
                        state <= new_address;
                else
                        bit_counter <= 7;
                        state <= ACK_init4;
                end if;

        ------ get ack
                when ACK_init4 =>
                clk_100 <= '0';
                SDA_1 <= '1';
                state <= ACK4;

                when ACK4 =>
                clk_100 <= '1';
                if      SDA = '0' then
                        bit_counter <= 7;
                        state <= receive_data; ---- go to receive
                            data from slave
                else
                        state <= error; ----- Ack-error
                end if;
```

```
--------- RECEIVE FROM THE SLAVE -------------
        when receive_data =>
        clk_100 <= '1';
        state <= receive_data1;

        when receive_data1 =>
        clk_100 <= '0';
        SDA_1 <= '1';
        state <= receive_data2;

        when receive_data2 =>
        clk_100 <= '1';
        shift_en <= '1';
        ---- First two bytes x register ----
        if byte_counter = 7 then
                state <= received_bytes1;
        elsif byte_counter = 15 then
                state <= received_bytes;
        ----- else: clock in more data -----
        else
                byte_counter <= byte_counter + 1;
                state <= receive_data1;
        end if;

        when received_bytes1 =>
--When received one byte of data, send ack bit.
        clk_100 <= '0';
        SDA_1 <= '0';
        byte_counter <= byte_counter + 1;
        state <= receive_data;

        when received_bytes =>
--When received one byte of data, send ack bit.
        clk_100 <= '0';
        SDA_1 <= '0';
        if data_ready = 0 then
                x_out_ina <= SensorData;
                --data_ready <= data_ready + 1;
                state <= nack;
        elsif data_ready = 1 then
                y_out_ina <= SensorData;
                --data_ready <= data_ready + 1;
```

```
                        state <= nack;
                    elsif data_ready = 2 then
                        z_out_ina <= SensorData;
                        data_ready <= 0;
                        state <= nack;
                    else
                        state <= error; --ERROR
                    end if;
                    byte_counter <= 0;



                    when nack => --Send Not Aknowledge bit,NACK(SCL is
                        high)
                    clk_100 <= '1';
                    state <= stop_init;

                --------- SEND STOP CONDITION -------------
                    ---- SDA goes from low to high, while SCL is high.
                    when stop_init => --prepare for the stop. set to
                        low
                    clk_100 <= '0';
                    SDA_1 <= '0';
                    state <= stop1;

                    when stop1 =>
                    clk_100 <= '1';
                    SDA_1 <= '0'; ----SDA goes from 0 to 1, while SCL
                        is 1-
                    state <= stop2;

                    when stop2 =>
                    clk_100 <= '1';
                    SDA_1 <= '1';
                    state <= idle;

                --------- ERROR -------------
                    when error =>
                    state <= idle;

                end case;
                end if;
        end if;
```

```
end process;

end I2C_Master_INA_arch;
```

B.4 Gyroscope I2C driver

```
-- Author      : Cecilie Bjelbole
-- Company     : University of Oslo
-- File name   : I2C_Master_ITG.vhd
-- Date        : 08.08.2012
-- Project     : ADCS for CubeSTAR
-- Function    : I2C MASTER DRIVER FOR THE ITG3200 GYRO

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity I2C_Master_ITG is
       port(
       SCL                  : out std_logic;
       SDA                  : inout std_logic;

       reset          : in std_logic;
       RegisterIn     : in std_logic_vector(7 downto 0);
       data_in        : in std_logic_vector(7 downto 0);
       temp_out       : out std_logic_vector(15 downto 0);
       x_out          : out std_logic_vector(15 downto 0);
       y_out          : out std_logic_vector(15 downto 0);
       z_out          : out std_logic_vector(15 downto 0);
       start_trans    : in std_logic;
       Read_Write     : in std_logic; --Read_Write = 1 when read
       clk_200        : in std_logic;

       busy           : out std_logic

       );
       end I2C_Master_ITG;
```

```
architecture I2C_Master_ITG_arch of I2C_Master_ITG is
type statetype is (idle, waiting, start, write_address, write_address2,
    ACK_init, ACK, send_reg_address, send_reg_address2, ACK_init2, ACK2,
    do_write, do_write2, ACK_init3, ACK3,
send_stop_init, send_stop, stop, do_read_init, start_1, start_2,
    new_address, new_address2, ACK_init4,
ACK4, receive_data, receive_data1, receive_data2, received_bytes1,
    received_bytes, nack, stop_init, stop1, stop2, error);

-------------------------SIGANL
    DECLARATIONS-----------------------------------

signal SDA_1                                    : std_logic;
signal clk_100                                  : std_logic;
signal shift_en                                 : std_logic;  --Trigger for
    the shift register
signal bit_counter                              : integer range 0 to 7:=7;
signal byte_counter                  : integer range 0 to 63:=0;
signal state                                    : statetype;
signal SensorData                               : std_logic_vector(15 downto
    0);
signal SensorData_in                 : std_logic_vector(15 downto 0);



------------------------------ CONSTANTS
    -------------------------------------
constant SlaveAddress_Read:  std_logic_vector(7 DOWNTO 0) :=
    "11010001";---- I2C address of the slave + read
constant SlaveAddress_Write:  std_logic_vector(7 DOWNTO 0) :=
    "11010000";---- I2C address of the slave + write
---"11010010" ITG3200 adress + write(0) P? breakout kortet.

 -------------------------------- Components
     ---------------------------------
 ------------------------------- Shift Register
     -------------------------------
component SR_SerIn_redge is
  generic (
    width : integer := 16);
  port
    (
```

```
    clk     : in std_logic;
    DataIn  : in std_logic;
    shift_en : in std_logic;
    DataOut : out std_logic_vector(width-1 downto 0)
    );
end component SR_SerIn_redge;


begin
----- Port Mapping ------

SR: entity work.SR_SerIn_redge(SR_SerIn_arch)
       port map(      clk => clk_100,
                                DataIn => SDA,
                                shift_en => shift_en,
                                DataOut => SensorData_in
                      );
                      SensorData <= SensorData_in(15 downto 0);



----- Set high to Z ------
SDA <= 'Z' when SDA_1 = '1' else '0';
SCL <= clk_100;

statm: process(clk_200)
begin
       if rising_edge(clk_200) then
       busy <= '1';
              if reset = '1' then
                     clk_100 <= '1'; ---Set SCL to clk_100
                     SDA_1 <= '1';
                     bit_counter <= 7; ---Start the bit counter at 7
                     byte_counter <= 0;
                     shift_en <= '0';
                     temp_out <= "0000000000000000";
                     x_out <= "0000000000000000";
                     y_out <= "0000000000000000";
                     z_out <= "0000000000000000";
                     state <= idle;
              else
                     shift_en <= '0';
              case state is
```

```
--------- IDLE MODE -------------
        when idle =>
        clk_100 <= '1';
        SDA_1 <= '1';
        busy <= '0';
        state <= waiting;

--------- WAITING ---------------
        when waiting =>
        clk_100 <= '1';
        SDA_1 <= '1';
        if start_trans = '1' then
                state <= start;
        else
                state <= waiting;
        end if;

--------- SENDING START CONDITION ----------
        when start => --- send start condition: Pull SCL
            high while SDA goes low
        clk_100 <= '1';
        SDA_1 <= '0';
        state <= write_address;

--------- WRITE SLAVE ADDRESS + write condition
    -----------
        when write_address => -- send 7 bit adress + R/W
            MSB sends first. SDA can only change when when
            SCL is at low
        clk_100 <= '0';
        SDA_1 <= SlaveAddress_Write(bit_counter); --Send
            the slave address
        state <= write_address2;

        when write_address2 =>
        clk_100 <= '1';
        if bit_counter > 0 then
                bit_counter <= bit_counter - 1;
                state <= write_address;
        else
                bit_counter <= 7;
                state <= ACK_init;
        end if;
```

```
-------- GET ACKNOWLEDGE FROM SLAVE ------------
        when ACK_init => --- Set SCL low and SDA high
            before
        --waiting for the ACK bit from the slave
        clk_100 <= '0';
        SDA_1 <= '1';
        state <= ACK;

        when ACK =>     --- Receiving ACK from slave
        clk_100 <= '1';
        if SDA <= '0' then
                state <= send_reg_address;
        else ------ ACK-error if SDA=1
                state <= error;
        end if;

--------- SEND REGISTER ADDRESS TO SLAVE ----------
        when send_reg_address =>
        clk_100 <= '0';
        --Register adress sent in with RegisterIn
        SDA_1 <= RegisterIn(bit_counter);
        state <= send_reg_address2;

        when send_reg_address2 => ---continue register
            adress sending
        clk_100 <= '1';
                if bit_counter > 0 then
                        bit_counter <= bit_counter -1;
                        state <= send_reg_address;
                else
                        bit_counter <= 7;
                        state <= ACK_init2;
                end if;

        ------ get ack
--- Set SCL low and SDA high before waiting for the ACK
    bit from the slave
        when ACK_init2 =>
        clk_100 <= '0';
        SDA_1 <= '1';
        state <= ACK2;
```

```
            when ACK2 =>   --- Receiving ACK from slave
            clk_100 <= '1';
            if SDA = '0' then
                    if Read_Write = '0' then ----Read_Write =
                        1 when read
                    state <= do_write;
                    else
                    state <= do_read_init; --- send repeated
                        start and then go do a read
                    end if;
            else state <= error; ----- ACK-error if SDA=1
            end if ;

        --------- WRITE TO THE SLAVE -------------
            when do_write =>
            clk_100 <= '0';
            SDA_1 <= data_in(bit_counter);
            state <= do_write2;

            when do_write2 =>
            clk_100 <= '1';
            if bit_counter > 0 then
                    bit_counter <= bit_counter -1;
                    state <= do_write;
            else
                    bit_counter <= 7;
                    state <= ACK_init3;
            end if;

            ------ get ack
         --- Set SCL low and SDA high before waiting for the ACK
            bit from the slave
            when ACK_init3=>
            clk_100 <= '0';
            SDA_1 <= '1';
            state <= ACK3;

            when ACK3 =>     --- Receiving ACK from slave
            clk_100 <= '1';
            if SDA = '0' then
                    state <= send_stop_init;
            else
                    state <= error;
```

```
            end if;



            ------ send stop
            when send_stop_init => ---- Prepare the stop
                condition. SDA and SCL is 0
            clk_100 <= '0';
            SDA_1 <= '0';
            state <= send_stop;

            when send_stop =>
            clk_100 <= '1';      ------ SCL is high for the
                stop
            SDA_1 <= '0';        ------ while the SDA goes
                from 0...
            state <= stop;

            when stop =>
            clk_100 <= '1';          ------ SCL remains high,
            SDA_1 <= '1';        ------ while SDA changes to
                high.
            state <= idle;       ------ both lines are at high
                = we are in idle.


        --------- READ FROM THE SLAVE -------------
        ----- need to send a repeated start signal after the
            address write.
            when do_read_init => ----- SDA starts at 1 to
                prepare for the 1 to 0 transition
            clk_100 <= '0';
            SDA_1 <= '1';
            state <= start_1;

            when start_1 =>
            clk_100 <= '1';
            SDA_1 <= '1';
            state <= start_2;

            when start_2 =>
            clk_100 <= '1';  ----- SCL stays at 1 while
            SDA_1 <= '0';    ----- SDA changes from high to low
```

```
                bit_counter <= 7;
                state <= new_address;

        ----- sending the slave address
                when new_address =>
                clk_100 <= '0';
                SDA_1 <= SlaveAddress_Read(bit_counter);
                state <= new_address2;

                when new_address2 =>
                clk_100 <= '1';
                if bit_counter > 0 then
                        bit_counter <= bit_counter - 1;
                        state <= new_address;
                else
                        bit_counter <= 7;
                        state <= ACK_init4;
                end if;

        ------ get ack
                when ACK_init4 =>
                clk_100 <= '0';
                SDA_1 <= '1';
                state <= ACK4;

                when ACK4 =>
                clk_100 <= '1';
                if      SDA = '0' then
                        bit_counter <= 7;
                        state <= receive_data; ---- go to receive
                            data from slave
                else
                        state <= error; ----- Ack-error
                end if;

        --------- RECEIVE FROM THE SLAVE -------------
                when receive_data =>
                clk_100 <= '1';
                state <= receive_data1;

                when receive_data1 =>
                clk_100 <= '0';
                SDA_1 <= '1';
```

```
state <= receive_data2;

when receive_data2 =>
clk_100 <= '1';
shift_en <= '1';
if byte_counter = 7 then
       state <= received_bytes1;
elsif byte_counter = 15 then
       state <= received_bytes; -- First two
           bytes is temp-register
elsif byte_counter = 23 then
       state <= received_bytes1;
elsif byte_counter = 31 then -- 3 and 4 is gyro x
    register
       state <= received_bytes;
elsif byte_counter = 39 then
       state <= received_bytes1; -- 5 and 6 is
           gyro y register
elsif byte_counter = 47 then
       state <= received_bytes;
elsif byte_counter = 55 then
       state <= received_bytes1; -- 7 and 8 is
           gyro z register
elsif byte_counter = 63 then
       state <= received_bytes;
else
       byte_counter <= byte_counter + 1;
       state <= receive_data1;
end if;

when received_bytes1 => --When received one byte
    of data, send ack bit.
clk_100 <= '0';
SDA_1 <= '0';
byte_counter <= byte_counter + 1;
state <= receive_data;

when received_bytes => --When received one byte of
    data, send ack bit.
clk_100 <= '0';
SDA_1 <= '0';
if byte_counter = 15 then
       temp_out <= SensorData;
```

```
                byte_counter <= byte_counter + 1;
                state <= receive_data;
            elsif byte_counter = 31 then
                x_out <= SensorData;
                byte_counter <= byte_counter + 1;
                state <= receive_data;
            elsif byte_counter = 47 then
                y_out <= SensorData;
                byte_counter <= byte_counter + 1;
                state <= receive_data;
            elsif byte_counter = 63 then
                z_out <= SensorData;
                byte_counter <= 0;
                state <= nack;
            else
                state <= error;
            end if;

            when nack =>    -----Send Not Aknowledge
                bit,NACK(SCL is high)
            clk_100 <= '1';
            state <= stop_init;
        --------- SEND STOP CONDITION -------------
            ---- SDA goes from low to high, while SCL is high.
            when stop_init =>  ----prepare for the stop. set
                to low
            clk_100 <= '0';
            SDA_1 <= '0';
            state <= stop1;

            when stop1 =>
            clk_100 <= '1';
            SDA_1 <= '0'; ----SDA goes from 0 to 1, while SCL
                is 1-
            state <= stop2;

            when stop2 =>
            clk_100 <= '1';
            SDA_1 <= '1';
            state <= idle;

            when error =>
            state <= waiting;
```

```
                end case;
                end if;
        end if;

end process;

end I2C_Master_ITG_arch;
```

B.5 ADCS Slave I2C driver

```
-- Author      : Cecilie Bjelbole
-- Co Author   : David Michael Bang
-- Company     : University of Oslo
-- File name   : I2C_Slave.vhd
-- Date        : 20.03.2012
-- Project     : ADCS for CubeSTAR
-- Function    : I2C SLAVE DRIVER


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity I2C_Slave is
        port (
                sclk                            : in     std_logic;
                clk_200                         : in std_logic;
                SCL                                     : in     std_logic;
                SDA                                     : inout std_logic;

                rst                                     : in    std_logic;
                system_rst                      : out   std_logic;
                housekeep_status   : in    std_logic_vector(7 downto 0);
                ADCS_en                         : out   std_logic;

                command                         : out std_logic_vector(7
                    downto 0);
```

```
                data_to_obc                 : in std_logic_vector(7
                    downto 0);
                busy                                : out std_logic;

                clk_enable                  : out std_logic;
                coils_enable            : out std_logic_vector(2 downto 0)
                );
end I2C_Slave;

architecture I2c_Slave_Arch of I2C_Slave is

type statetype is (init, waiting,wait_start,init_read_address,
    read_address,
init_ADD_CHK , ADD_CHK, read_ACK, write_ACK, command_ACK, coil_ACK,
receiving_ACK, init_wait_command, init_read_command, read_command,
init_wait_coilcommand, init_read_coilcommand, read_coilcommand,
    COIL_CHK,
CMD_CHK, wait_send_data, wait_send_data2,
send_data, sending_data, send_data2, error);

---------------------------
--- SIGNAL DECLARATIONS ---
---------------------------

------------------------------ I2C DATA SIGNALS
    -------------------------------

signal SDA_out                      : std_logic;
signal SDA_signal_out       : std_logic;
signal SDA_signal_in        : std_logic;

------------------------------- CLOCK SIGNALS
    ---------------------------------

signal i2c_clk                      : std_logic_vector (1 downto 0);
signal i2c_sda                      : std_logic_vector (1 downto 0);
signal clk_100                      : std_logic;
signal clk_200_test                 : std_logic;

--------------------------- TRIGGER/ENABLE SIGNALS
    ---------------------------

signal SDA_out_enable       : std_logic;
```

```
signal Start_cond                   : std_logic;
signal Stop_cond                    : std_logic;
signal shift_en                     : std_logic;
signal bit_inc                      : std_logic;
signal Check_Start_cond     : std_logic;
signal load_en                      : std_logic;
signal data_loaded                  : std_logic;
signal scl_rise                     : std_logic;
signal scl_fall                     : std_logic;
signal sda_rise                     : std_logic;
signal sda_fall                     : std_logic;
signal stop                               : std_logic;
signal par_en                       : std_logic;
signal coil                               : integer range 4 downto 0;
signal inc                                : std_logic;
signal inc_j                        : std_logic;
-------------------------------- DATA VECTOR
    ----------------------------------

signal PDATA                        : std_logic_vector(7 downto 0);
signal Sdata                        : std_logic;
signal response_data        : std_logic_vector(7 downto 0);
signal housekeep                    : std_logic_vector(7 downto 0);

------------------ Bit counters for sending and receiving data
    ----------------

signal bit_count                    : integer range 7 downto 0;
signal i                                  : integer range 5 downto 0;
signal j                                  : integer range 2 downto 0;

----------------------------------- DIV
    ---------------------------------------

signal state                        : statetype;
signal rst_i                        : std_logic;
signal locked                       : std_logic;
signal Command_Number       : integer range 30 downto 0;

-----------------------------
--- CONSTANT DECLARATIONS ---
-----------------------------
```

```
---------------------------- I2C SLAVE ADDRESS + R/W
    ----------------------------
---------------------- The ADCS I2C slave address is 0x10
    ------------------------
constant I2C_slave_address_write    : std_logic_vector(7 downto 0) :=
    "00100000";
constant I2C_slave_address_read     : std_logic_vector(7 downto 0) :=
    "00100001";

------------------------------ I2C SLAVE COMMANDS
    --------------------------------

--ADCS STATUS 0x1E
constant I2C_command_1       : std_logic_vector(7 downto 0) :=
    "00011110";

--ADCS DIAGNOSTIC 0x2D
constant I2C_command_2       : std_logic_vector(7 downto 0) :=
    "00101101";

--ADCS RESTART 0x55
constant I2C_command_3       : std_logic_vector(7 downto 0) :=
    "01010101";

--ADCS ACTIVATE MAGNETIC COILS 0x87
constant I2C_command_4       : std_logic_vector(7 downto 0) :=
    "10000111";
constant coil_1                     : std_logic_vector(7 downto 0) :=
    "00000001";
constant coil_2                     : std_logic_vector(7 downto 0) :=
    "00000010";
constant coil_3                     : std_logic_vector(7 downto 0) :=
    "00000011";

--ADCS ENABLE 0x99
constant I2C_command_5       : std_logic_vector(7 downto 0) :=
    "10011001";

--ADCS DISABLE 0xAA
constant I2C_command_6       : std_logic_vector(7 downto 0) :=
    "10101010";

--ADCS GET ATTITUDE
```

```
constant I2C_command_7       : std_logic_vector(7 downto 0) :=
    "10110100";


------------------------------
--- COMPONENT DECLARATIONS ---
------------------------------

--- SERIAL IN PARALLEL OUT SHIFT REGISTER ---
component SR_SerIn_redge8 is
  generic (
    width : integer := 8);
  port
    (
    clk     : in  std_logic;
    DataIn  : in  std_logic;
    shift_en : in std_logic;
    DataOut : out std_logic_vector(width-1 downto 0)
    );
end component SR_SerIn_redge8;

--- PARALLELL IN SERIAL OUT SHIFT REGISTER ---
component SR_ParIn_redge8 is
  generic (
    width : integer := 8);
port
    (
    clk                     : in  std_logic;
    data_in                 : in  std_logic_vector(width-1 downto 0);
    load_en                 : in  std_logic;
         shift                      : in   std_logic;
         data_loaded        : out std_logic;
    data_out                : out std_logic
    );
end component SR_ParIn_redge8;



begin

--------------------
--- PORT MAPPING ---
--------------------
```

```
BUFF: SR_SerIn_redge8
       port map(sclk, SDA_signal_in, shift_en, PDATA);
--sclk???
PAR: SR_ParIn_redge
       port map(sclk, response_data, load_en, par_en, data_loaded,
           Sdata);

-----------------
--- PROCESSES ---
-----------------

--- INCREASE "BIT_count" BY 1 IF "BIT_inc" = 1 ---
BITINC: process(sclk)
begin
       if rising_edge(sclk) then
              if rst_i = '1' then
                     bit_count <= 0;
              elsif bit_inc = '1' then
                     if  bit_count = 7 then
                            bit_count <= 0;
                     else
                            bit_count <= bit_count + 1;
                     end if;
              end if;
       end if;
end process BITINC;

COUNTING: process(sclk)
begin
       if rising_edge(sclk) then
              if rst_i = '1' then
                     i <= 0;
                     j <= 0;
              elsif inc = '1' then
                     if  i = 5 then
                            i <= 0;
                     else
                            i <= i + 1;
                     end if;
              elsif inc_j = '1' then
                     if  j = 2 then
                            i <= 0;
```

```
                        else
                                i <= i + 1;
                        end if;
                end if;
        end if;
end process COUNTING;


--- Process for enabling the PLL clock ---
Start_pll_clock: process(SCL)
begin
        if falling_edge(SDA) then
                clk_enable <= '1';
        end if;
end process Start_pll_clock;


Control: process(sclk)
begin --Checking rising and falling edge for I2C clk sync
        --The fsm stay in the current state until a rise or fall
        --on the clock line is detected.
        if rising_edge(sclk) then
                i2c_clk(0) <= SCL;
                i2c_clk(1) <= i2c_clk(0);

                scl_fall <= not i2c_clk(0) and i2c_clk(1);
                scl_rise <= i2c_clk(0) and not i2c_clk(1);

                i2c_sda(0) <= SDA_signal_in;
                i2c_sda(1) <= i2c_sda(0);

                sda_fall <= not i2c_sda(0) and i2c_SDA(1);
                sda_rise <= i2c_sda(0) and not i2c_SDA(1);

                if sda_rise = '1' and SCL = '1' then
                        stop <= '1';
                else
                        stop <= '0';
                end if;

        end if;
end process Control;
```

```
--- PROCESS FOR CHECKING THE I2C START_CONDITION ---

--- THE PROCESS "Chk_Start" CHECKS IF BOTH "clk_100" AND SDA_SIGNAL IN
    IS '1'
--- IF TRUE THEN "Check_Start_cond <= '1'"
Chk_Start: process(sclk)
begin
        if rising_edge(sclk) then
                if clk_100 = '1' and SDA_signal_in = '1' then
                        Check_Start_cond <= '1';
                else
                        Check_Start_cond <= '0';
                end if;
        end if;
end process Chk_Start;

--- THE PROCESS "Start_I2C" CHECKS IF "Check_Start_cond" = '1'
--- IF TRUE THEN IT CHECKS IF "clk_100" = '1' AND "SDA_signal_in" = '0'
Start_I2C: process(sclk)
begin
        if rising_edge(sclk) then
                start_cond <= '0';
                if Check_start_cond = '1' then
                        if SDA_signal_in = '0' and clk_100 = '1' then
                                Start_cond <= '1';
                        else
                                Start_cond <= '0';

                        end if;
                end if;
        end if;
end process Start_I2C;
-------------------------------
-------------------------------
clk_200_test <= clk_100;
clk_100 <= SCL;
SDA_signal_in <= SDA;
SDA_out <= 'Z' when SDA_signal_out = '1' else '0';
SDA <= SDA_out when SDA_out_enable = '1' else 'Z';
response_data <= data_to_obc;
----------------------------
--- FINITE STATE MACHINE ---
```

```
----------------------------
FSM: process(rst,clk_100,sclk,stop) begin
       if rst = '1' then
              command <= "00000000";
              rst_i <= '1';
              state <= init;
       elsif stop = '1' then
              state <= init;
       elsif rising_edge(sclk) then
              shift_en <= '0';
              load_en <= '0';
              par_en <= '0';
              bit_inc <= '0';
              inc <= '0';
              SDA_out_enable <= '0';
              SDA_signal_out <= '1';
              rst_i <= '0';
              case state is
                     -------------------
                     -- Initial state --
                     -------------------
                     when init =>
                            rst_i <= '1';
                            busy <= '0';
                            Command_Number <= 0;
                            if stop_cond = '1' then
                                   state <= init;
                            else
                     --
                                   state <= waiting;
                            end if;

                     -------------------------------------
                     -- Waiting for I2C start condition --
                     -------------------------------------
                     when waiting =>
                            if start_cond = '1' then
                                   state <= wait_start;
                            else
                                   state <= waiting;
                            end if;

                     when wait_start =>
```

```
busy <= '1';
        if scl_rise = '1' then
                --shift_en <= '1';
                --bit_inc <= '1';
                state <= init_read_address;
        else
        --stay here for clk sync
                state <= wait_start;
        end if;
--------------------------------
-- Reading I2C slave address --
--------------------------------

-- SCL = '0'
when init_read_address =>
if scl_fall = '1' then
        shift_en <= '1';
        bit_inc <= '1';
        if bit_count = 7 then
                        state <= init_ADD_CHK;
                elsif bit_count < 7 then
                        state <= read_address;
                else
                        state <= error;
                end if;
else
        state <= init_read_address;
end if;



when read_address =>
        if scl_rise = '1' then
                state <= init_read_address;
        else
                state <= read_address;
        end if;

--------------------------------
-- Checking I2C slave address --
--------------------------------

-- SCL = '0'
```

```
when init_ADD_CHK =>
       state <= ADD_CHK;

-- SCL = '0'
when ADD_CHK =>

              if PDATA = I2C_slave_address_write
                  then
                      SDA_out_enable <= '1'; --
                      SDA_signal_out <= '0'; --
                      if scl_rise = '1' then
                             state <= write_ACK;
                      else
                             state <= ADD_CHK;
                      end if;

              elsif PDATA =
                  I2C_slave_address_read then
                      SDA_out_enable <= '1';
                      SDA_signal_out <= '0';
                      if scl_rise = '1' then
                             state <= read_ACK;
                      else
                             state <= ADD_CHK;
                      end if;
              else
                      state <= error;
              end if;



    --------------------------
-- Send acknowledge bit --
    --------------------------

    -- SCL = '0'
    when write_ACK =>
           SDA_out_enable <= '1';
      SDA_signal_out <= '0';
           if scl_fall = '1' then
                  state <= init_wait_command;
           else
                  state <= write_ACK;
```

```
                end if;

        -- SCL = '0'
    when read_ACK =>
                SDA_out_enable <= '1';
           SDA_signal_out <= '0';
                if scl_fall = '1' then
                        state <= send_data;
                else
                        state <= read_ACK;
                end if;


        -- SCL = '0'
        when command_ACK =>
                SDA_out_enable <= '1';
                SDA_signal_out <= '0';
                if scl_fall = '1' then
                        if Command_Number = 4 then
                                state <=
                                    init_wait_coilcommand;
                                    --- to determine wich coil
                        else
                                state <= waiting;
                        end if;
                else
                        state <= command_ACK;
                end if;

        when coil_ACK =>
                SDA_out_enable <= '1';
           SDA_signal_out <= '0';
                if scl_fall = '1' then
                        state <= waiting;
                else
                        state <= coil_ACK;
                end if;
        ------------------------------
        -- Receiving acknowledge bit --
        ------------------------------
                -- SCL = '0'
    when receiving_ACK =>
                if scl_fall = '1' then
```

```
                        if SDA = '0' then
                                if i = 0 and j = 0 then
                                        state <= init;
                                elsif i > 0 then
                                        state <= send_data;
                                elsif j > 0 then
                                        state <= send_data;
                                end if;
                        else
                                state <= send_data;--error;
                        end if;
                else
                        state <= error;
                end if;


        ---------------------------------
        -- Reading I2C command address --
        ---------------------------------
        when init_wait_command =>
                if scl_rise = '1' then
                        state <= init_read_command;
                else
                        state <= init_wait_command;
                end if;

        -- SCL = '1'
        when init_read_command =>
                if scl_fall = '1' then
                        shift_en <= '1';
                        bit_inc <= '1';
                        if bit_count = 7 then
                                state <= CMD_CHK;
                        elsif bit_count < 7 then
                                state <= read_command;
                        else
                                state <= error;
                        end if;
                else
                        state <= init_read_command;
                end if;

        -- SCL = '0'
```

```
when read_command =>
        if scl_rise = '1' then
                state <= init_read_command;
        else
                state <= read_command;
        end if;

---------------------------
-- Reading coil command --
---------------------------
when init_wait_coilcommand =>
        if scl_rise = '1' then
                state <= init_read_coilcommand;
        else
                state <= init_wait_coilcommand;
        end if;

when init_read_coilcommand =>
        if scl_fall = '1' then
                shift_en <= '1';
                bit_inc <= '1';
                if bit_count = 7 then
                        state <= COIL_CHK;
                elsif bit_count < 7 then
                        state <= read_coilcommand;
                else
                        state <= error;
                end if;
        else
                state <= init_read_coilcommand;
        end if;

-- SCL = '0'
when read_coilcommand =>
        if scl_rise = '1' then
                state <= init_read_coilcommand;
        else
                state <= read_coilcommand;
        end if;

---------------------------
-- Cheking coil command --
---------------------------
```

```
when COIL_CHK =>
SDA_out_enable <= '1';
SDA_signal_out <= '0';
if scl_rise = '1' then
        ---- Value 8 = coil 1 = coil X
        if PDATA = "10000111" then --0x87
                coils_enable <= "001";
                coil <= 1;
                state <= coil_ACK;
        ---- Value 4 = coil 2 = coil Y
        elsif PDATA = "01001011" then --0x4B
                coils_enable <= "010";
                coil <= 2;
                state <= coil_ACK;
        ---- Value 6 = coil 3 = coil Z
        elsif PDATA = "00101101" then --0x2D
                coils_enable <= "011";
                coil <= 3;
                state <= coil_ACK;
                ---- Value 0 - Turn off
        elsif PDATA = "00000000" then --0x0
                coils_enable <= "000";
                coil <= 0;
                state <= coil_ACK;
        else
                state <= error;
        end if;
else
        state <= COIL_CHK;
end if;

--------------------------------------
-- Checking/Reading command address --
--------------------------------------

-- SCL = '1'
when CMD_CHK =>

        SDA_out_enable <= '1';
        SDA_signal_out <= '0';
if scl_rise = '1' then
                ---- STATUS ----
        if PDATA = I2C_command_1 then
```

```
                command <= "00011110";
                Command_Number <= 1;
                state <= command_ACK;

                ---- DIAGNOSTIC ----
        elsif PDATA = I2C_command_2 then
                Command_Number <= 2;
                command <= "00101101";
                state <= command_ACK;

                ---- RESTART ----
        elsif PDATA = I2C_command_3 then
                command <= "01010101";
                Command_Number <= 3;
                state <= command_ACK;

                ---- ACTIVATE COILS ----
        elsif PDATA = I2C_command_4 then
                command <= "10000111";
                Command_Number <= 4;
                state <= command_ACK;

                ---- ENABLE ----
        elsif PDATA = I2C_command_5 then
                command <= "10011001";
                Command_Number <= 5;
                state <= command_ACK;

                ---- DISABLE ----
        elsif PDATA = I2C_command_6 then
                command <= "10101010";
                Command_Number <= 6;
                state <= command_ACK;

                ---- GET ATTITUDE ----
        elsif PDATA = I2C_command_7 then
                command <= "10110100";
                Command_Number <= 7;
                state <= command_ACK;

        else
                state <= error;
        end if;
```

```
else
       state <= CMD_CHK;
end if;



-------------------------
-- Send data to master --
-------------------------
when send_data =>
busy <= '0';
SDA_out_enable <= '1';
SDA_signal_out <= Sdata;

       if Command_Number = 0 then
              state <= init;

       elsif Command_Number = 1 then
 --Returning the corresponding command
              --response_data <= data_to_obc;

              state <= wait_send_data;

       elsif Command_Number = 2 then

              inc <= '1';
              state <= wait_send_data;


       elsif Command_Number = 3 then
              --system_rst <= '1';
              --rst_i <= '1';
              state <= wait_send_data;
              --state <= init;


       elsif Command_Number = 4 then
              if coil = 1 then

                     state <= wait_send_data;
              elsif coil = 2 then

                     state <= wait_send_data;
```

```
                elsif coil = 3 then

                        state <= wait_send_data;
                else
                        state <= error;
                end if;

        elsif Command_Number = 5 then
                ADCS_en <= '1';
                --response_data <=
                    "10011001";--ADCS_mode;

                inc_j <= '1';
                state <= wait_send_data;

        elsif Command_Number = 6 then
                ADCS_en <= '0';
                --response_data <=
                    "10101010";--ADCS_mode;

                state <= wait_send_data;

        else
                state <= error;
        end if;

when wait_send_data =>
        load_en <= '1';
        par_en <= '1';
        state <= wait_send_data2;
when wait_send_data2 =>
        busy <= '1';
        if scl_rise = '1' then
                state <= sending_data;
        else
                state <= wait_send_data2;
        end if;

when sending_data =>
busy <= '1';
SDA_out_enable <= '1';
SDA_signal_out <= Sdata;
if scl_fall = '1' then
```

```
                        par_en <= '1';
                        bit_inc <= '1';
                        if bit_count = 7 then
                                state <= receiving_ACK;
                        elsif bit_count < 7 then
                                state <= send_data2;
                                --state <= wait_send_data;
                        else
                                state <= error;
                        end if;
                else
                        state <= sending_data;
                end if;


                when send_data2 =>
                        if scl_rise = '1' then
                                SDA_out_enable <= '1';
                                SDA_signal_out <= Sdata;
                                state <= sending_data;
                        else
                                state <= send_data2;
                        end if;


                -----------
                -- ERROR --
                -----------
                when error =>
                        state <= init;

        end case;
    end if;

end process FSM;
end I2c_Slave_Arch;
```

B.6 Shift Register

```
-- Author      : Cecilie Bjelbøle
-- Company     : University of Oslo
```

```
-- File name   : SR_SerIn_redge8.vhd
-- Date        : 28.08.2012
-- Project     : ADCS CubeSTAR
-- Function    : Serial in parallel out shift register.



library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity SR_SerIn_redge8 is
  generic (
    width : integer := 8);


  port
    (
    clk     : in  std_logic;
    DataIn  : in  std_logic;
    shift_en : in std_logic;
    DataOut : out std_logic_vector(width-1 downto 0)
    );

end SR_SerIn_redge8;


architecture SR_SerIn8_arch of SR_SerIn_redge8 is

  signal data_int : std_logic_vector(width-1 downto 0);


begin

  SHIFT_REG:
  process (clk)
  begin

   if rising_edge(clk) then
      if (shift_en = '1') then
        data_int(0) <= DataIn;
```

```
        for i in width-2 downto 0 loop
          data_int(i+1) <= data_int(i);
        end loop;
      end if;

    end if;
  end process SHIFT_REG;

  DataOut <= data_int;


end SR_SerIn8_arch;
```

```
-- Author      : Cecilie Bjelboele
-- Company     : University of Oslo
-- File name   : SR_ParIn_redge.vhd
-- Date        : 06.09.2012
-- Project     : ADCS for CubeSTAR
-- Function    : Parallel in serial out shift register.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity SR_ParIn_redge is
  generic (
    width : integer := 8);

        port
    (
    clk               : in  std_logic;
    data_in           : in  std_logic_vector(width-1 downto 0);
    load_en           : in  std_logic;
         shift                : in  std_logic;
         data_loaded  : out std_logic;
    data_out          : out std_logic
    );
end SR_ParIn_redge;

architecture SR_ParIn_redge_arch of SR_ParIn_redge is

        signal temp    : std_logic_vector(width-1 downto 0);
```

```
        signal int_out : std_logic;

begin

REG:
process(clk)
        begin

   if rising_edge(clk) and shift = '1' then

                if (load_en = '1') then
                        temp <= data_in(width-1 downto 0);
                        data_loaded <= '1';
                else
                        data_loaded <= '0';
                        for i in width-2 downto 0 loop
                                temp(i+1) <= temp(i);
                        end loop;

                end if;

        end if;

end process REG;
   data_out <= temp(width-1);

end architecture;
```

B.7 PWM module

```
-- Author      : Cecilie Bjelbole
-- Company     : University of Oslo
-- File name   : PWM.vhd
-- Date        : 28.08.2012
-- Project     : ADCS for CubeSTAR
-- Function    : Pulse widht modulation for the coil drivers
-- Description : The coil, direction of the current and the
--                              duty cycle of the PWM signal are given by
--                              enable vector.
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity PWM is
	port(
	clk			: in std_logic;

	x_out1			: out std_logic;
	x_out2			: out std_logic;
	y_out1			: out std_logic;
	y_out2			: out std_logic;
	z_out1			: out std_logic;
	z_out2			: out std_logic;

-- PWM enable 001/010 = X-axis, 011/100 = Y-axis, 101/110 = Z-axis
	enable		: in std_logic_vector(5 downto 0)

	);
	end PWM;

architecture PWM_arch of PWM is

signal pwm_width : integer range 0 to 800;
signal i : integer range 0 to 800;
-- 20 MHz / 800 = 25 kHz
-- 20 MHz / 200 = 100 kHz
-- 20MHz / 950 = 21 kHz

begin
CNT: process(clk)
begin
	if rising_edge(clk) then
		if enable(5 downto 3) = "000" then
			i <= 0;
		else
			if  i = 800 then
				i <= 0;
			else
				i <= i + 1;
			end if;
		end if;
	end if;
```

```
end process CNT;
WidthCNT: process(clk)
begin
        if rising_edge(clk) then
                if enable(2 downto 0) = "000" then
                        pwm_width <= 700;
                elsif enable (2 downto 0) = "001" then
                        pwm_width <= 600;
                elsif enable (2 downto 0) = "010" then
                        pwm_width <= 500;
                elsif enable (2 downto 0) = "011" then
                        pwm_width <= 400;
                elsif enable (2 downto 0) = "100" then
                        pwm_width <= 300;
                elsif enable (2 downto 0) = "101" then
                        pwm_width <= 200;
                elsif enable (2 downto 0) = "110" then
                        pwm_width <= 100;
                elsif enable (2 downto 0) = "111" then
                        pwm_width <= 50;
                end if;
        end if;
end process WidthCNT;


statm: process(clk)
begin
        if rising_edge(clk) then
                if enable(5 downto 3) = "000" then
                --Break-mode:
                x_out1 <= '1';
                x_out2 <= '1';
                y_out2 <= '1';
                y_out1 <= '1';
                z_out1 <= '1';
                z_out2 <= '1';

                else
                --------- start counting -------------
                        --- X AXIS ---
                if enable(5 downto 3) = "001" then
                x_out2 <= '0';
                y_out2 <= '1';
```

```
y_out2 <= '1';
z_out1 <= '1';
z_out2 <= '1';
        if i < pwm_width then
                x_out1 <= '0';
         elsif i = pwm_width then
                x_out1 <= '1';
         elsif i = 800 then
                x_out1 <= '0';
         end if;
-- To control the H bridge in two directions:
elsif enable(5 downto 3) = "010" then
x_out1 <= '0';
y_out2 <= '1';
y_out2 <= '1';
z_out1 <= '1';
z_out2 <= '1';
        if i < pwm_width then
                x_out2 <= '0';
        elsif i = pwm_width then
                x_out2 <= '1';
        elsif i = 800 then
                x_out2 <= '0';
        end if;

        --- Y AXIS ---
elsif enable(5 downto 3) = "011" then
x_out1 <= '1';
x_out2 <= '1';
y_out2 <= '0';
z_out1 <= '1';
z_out2 <= '1';
        if i < pwm_width then
                y_out1 <= '0';
        elsif i = pwm_width then
                y_out1 <= '1';
        elsif i = 800 then
                y_out1 <= '0';
        end if;
elsif enable(5 downto 3) = "100" then
x_out1 <= '1';
x_out2 <= '1';
y_out1 <= '0';
```

```
z_out1 <= '1';
z_out2 <= '1';
       if i < pwm_width then
              y_out2 <= '0';
       elsif i = pwm_width then
              y_out2 <= '1';
       elsif i = 800 then
              y_out2 <= '0';
       end if;

        --- Z AXIS ---
elsif enable(5 downto 3) = "101" then
x_out1 <= '1';
x_out2 <= '1';
y_out1 <= '1';
y_out2 <= '1';
z_out2 <= '1';
       if i < pwm_width then
              z_out1 <= '0';
       elsif i = pwm_width then
              z_out1 <= '1';
       elsif i = 800 then
              z_out1 <= '0';
       end if;
elsif enable(5 downto 3) = "110" then
x_out1 <= '1';
x_out2 <= '1';
y_out1 <= '1';
y_out2 <= '1';
z_out1 <= '1';
              if i < pwm_width then
              z_out2 <= '0';
        elsif i = pwm_width then
              z_out2 <= '1';
       elsif i = 800 then
              z_out2 <= '0';
        end if;
elsif enable(5 downto 3) = "111" then
--Break-mode:
x_out1 <= '1';
x_out2 <= '1';
y_out1 <= '1';
y_out2 <= '1';
```

```
                z_out2 <= '1';


                end if;
                end if;
                end if;

end process;

end PWM_arch;
```

Appendix C

C Code

```
/*********************************************************
*
*  File:   OBCcontrol.c
*  Project: CubeSTAR ADCS card version 2
*  Author: Cecilie Bjelbøle
*  Revised: December 2012
*
*********************************************************/
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <float.h>
#include <unistd.h>

#include "main_ADCS_CB.h"

#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_uart.h"
#include "altera_avalon_uart_regs.h"
#include "alt_types.h"

char instruction;
char inst;
int start_obc;
volatile int busy_slave;
uint16_t response;
void initgyro(void){
```

```
        IOWR_ALTERA_AVALON_UART_TXDATA(UART_0_BASE, instruction);
//Initialize the Gyroscope

        IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,0);
        IOWR_ALTERA_AVALON_PIO_DATA(START_TRANS_ITG_BASE,1);
        IOWR_ALTERA_AVALON_PIO_DATA(READ_WRITE_BASE,0);//Write
// Configure Interrupt Register, REG 0x17
        IOWR_ALTERA_AVALON_PIO_DATA(REG_ITG_BASE,0x17);
        //Enables active high level, push-pull, latch until interrupt
            cleared, any register read,
        //enable intr when device is ready = 0x34
        IOWR_ALTERA_AVALON_PIO_DATA(GYRO_DATA_BASE,0x34);

        while(IORD(BUSY_ITG_BASE,0)); //Wait until transfer is complete
// Set sample rate divider register
        IOWR_ALTERA_AVALON_PIO_DATA(REG_ITG_BASE,0x15);
        IOWR_ALTERA_AVALON_PIO_DATA(GYRO_DATA_BASE,0x01);

        while(IORD(BUSY_ITG_BASE,0)); //Wait until transfer is complete
 // Set Full scale range & 42Hz LP filter
        IOWR_ALTERA_AVALON_PIO_DATA(REG_ITG_BASE,0x16);
        IOWR_ALTERA_AVALON_PIO_DATA(GYRO_DATA_BASE,0x1B);
        while(IORD(BUSY_ITG_BASE,0)); //Wait until transfer is complete
//stop transaction
//IOWR_ALTERA_AVALON_PIO_DATA(START_TRANS_ITG_BASE,0);
}

void initmagn(){
//Initialize the Magnetometer
        //start write transaction
        IOWR_ALTERA_AVALON_PIO_DATA(START_MAG_BASE, 1);
        IOWR_ALTERA_AVALON_PIO_DATA(READ_WRITE_HMC_BASE,0);
        //Make sure reset off:
        IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,0);

        //Configuration Register A
        //Reg = 0x00;
        IOWR_ALTERA_AVALON_PIO_DATA(REGISTER_HMC_BASE, 00);
        //Config = 0x58;
        IOWR_ALTERA_AVALON_PIO_DATA(DATA_HMC_BASE, 0x58);


while(IORD(BUSY_HMC_BASE,0)); //Wait until transfer is complete
```

```
        //Mode Register
        //Configure the register to Continious Measurement Mode
        //for single measurement Reg = 0x01
        //Reg = 0x02;
        IOWR_ALTERA_AVALON_PIO_DATA(REGISTER_HMC_BASE, 0x02);
        //Config = 0x00;
        IOWR_ALTERA_AVALON_PIO_DATA(DATA_HMC_BASE, 00);
        while(IORD(BUSY_HMC_BASE,0)); //Wait until transfer is complete

         //IOWR_ALTERA_AVALON_PIO_DATA(START_MAG_BASE,0);
}

int magnetometer(void){

while(IORD(BUSY_HMC_BASE,0)); //Wait until transfer is complete
        //Start Read transaction
        IOWR_ALTERA_AVALON_PIO_DATA(START_MAG_BASE,1);
        IOWR_ALTERA_AVALON_PIO_DATA(READ_WRITE_HMC_BASE,1);
        //Make sure reset off:
        IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,0);

        // Set adress pointer to dataregister
        IOWR_ALTERA_AVALON_PIO_DATA(REGISTER_HMC_BASE, 3);

        magnetometer_x = IORD_ALTERA_AVALON_PIO_DATA(MAG_X_BASE);
        mag_x = magnetometer_x;
        magnetometer_y = IORD_ALTERA_AVALON_PIO_DATA(MAG_Y_BASE);
        mag_y = magnetometer_y;
        magnetometer_z = IORD_ALTERA_AVALON_PIO_DATA(MAG_Z_BASE);
        mag_z = magnetometer_z;


        printf("HMC:%04x%04x%04x\n", mag_x, mag_y, mag_z);
        return 0;
}

void initina(void){
//Initialize the INA Current Monitor
//the registers in INA is volatile and needs to be reconfigured after
    each power up.
        //Start transaction Write
```

```
        IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,0);//make sure reset
            = 0
        IOWR_ALTERA_AVALON_PIO_DATA(START_INA_BASE, 1);
        IOWR_ALTERA_AVALON_PIO_DATA(READ_WRITE_INA_BASE, 0);

        //The calibration register :
        IOWR_ALTERA_AVALON_PIO_DATA(REGISTER_INA_BASE, 0x05);
        //max current of 80mA = 3uA/bit resolution:
        IOWR_ALTERA_AVALON_PIO_DATA(INA226_DATA_BASE, 0x130C);

}

void gyro(void){
        IOWR_ALTERA_AVALON_UART_TXDATA(UART_0_BASE, instruction);

        //Start Read transaction
        IOWR_ALTERA_AVALON_PIO_DATA(START_TRANS_ITG_BASE,1);
        IOWR_ALTERA_AVALON_PIO_DATA(READ_WRITE_BASE,1);

        // Set adress pointer to dataregister
        IOWR_ALTERA_AVALON_PIO_DATA(REG_ITG_BASE,0x1B);
        //Read out data
        temperature = IORD_ALTERA_AVALON_PIO_DATA(GYRO_TEMP_BASE);
        temp = temperature;
        gyro_x = IORD_ALTERA_AVALON_PIO_DATA(GYRO_X_BASE);
        gyro_x_out = gyro_x;
        gyro_y = IORD_ALTERA_AVALON_PIO_DATA(GYRO_Y_BASE);
        gyro_y_out = gyro_y;
        gyro_z = IORD_ALTERA_AVALON_PIO_DATA(GYRO_Z_BASE);
        gyro_z_out = gyro_z;

        printf("ITG:%04x%04x%04x%04x\n", temp, gyro_x_out, gyro_y_out,
            gyro_z_out);
}

void current(void){
        //Start read transaction
        IOWR_ALTERA_AVALON_PIO_DATA(READ_WRITE_INA_BASE, 1);

        // Set address pointer to data register
        IOWR_ALTERA_AVALON_PIO_DATA(REGISTER_INA_BASE, 0x04);
        usleep(200);
        current_x = IORD_ALTERA_AVALON_PIO_DATA(INA_X_BASE);
```

```
        current_y = IORD_ALTERA_AVALON_PIO_DATA(INA_Y_BASE);
        current_z = IORD_ALTERA_AVALON_PIO_DATA(INA_Z_BASE);
        //send current through a coil.
        //IOWR_ALTERA_AVALON_PIO_DATA(COILS_ENABLE_BASE, 0x3);//Y coil

        //print data
        printf("%04x%04x%04x\n", current_x, current_y, current_z);

}


// *************      MAIN       ***************** //
int main () {
  // Main function initiates and stars the instruction handler //
        printf("Welcome to OBCcontrol!\n");
        IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,1);
        IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,0);

  // *** INSTRUCTION HANDLER ***
    //Initialize the sensors:
        initina();
        initgyro();
        initmagn();
while(1){

        //current();
                //IOWR_ALTERA_AVALON_PIO_DATA(COILS_ENABLE_BASE, 0xF);


        command = IORD(COMMAND_BASE,0);
        //command = IORD_ALTERA_AVALON_PIO_DATA(COMMAND_BASE);
        busy_slave = IORD(BUSY_SLAVE_BASE,0);
if ( command != ' ' ){
//Several of these commands are not fully implemented because of the
    lack of
//an entire ADCS. Several cases returns the corresponding command.
        switch( command ){
        //status
                case 0x1E: //Sends the given command in return
                        response = 0x1E;
                        IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                            response);
                break;
```

```
//diagnostics
        case 0x2D:
                magnetometer();
                gyro();
                current();
        //send magnetometer data
                response = IORD(MAG_X_BASE,0);
                uint8_t LOWmag_x = (response & 0xff); //get 8 LSB
                uint8_t HIGHmag_x = ((response>>8) & 0xff);
                    //right shift 8 places and get 8 LSB
                IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                    HIGHmag_x);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete
                IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                    LOWmag_x);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete
                response = IORD(MAG_Y_BASE,0);
                uint8_t LOWmag_y = (response & 0xff); //get 8 LSB
                uint8_t HIGHmag_y = ((response>>8) & 0xff);
                    //right shift 8 places and get 8 LSB
                IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                    HIGHmag_y);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete
                IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                    LOWmag_y);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete
                response = IORD(MAG_Z_BASE,0);
                uint8_t LOWmag_z = (response & 0xff); //get 8 LSB
                uint8_t HIGHmag_z = ((response>>8) & 0xff);
                    //right shift 8 places and get 8 LSB
                IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                    HIGHmag_z);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete
                IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                    LOWmag_z);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete
```

```
//send gyro data
        response = IORD(GYRO_TEMP_BASE,0);
        uint8_t LOWgyro_temp = (response & 0xff); //get 8
            LSB
        uint8_t HIGHgyro_temp = ((response>>8) & 0xff);
            //right shift 8 places and get 8 LSB
        IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
            HIGHgyro_temp);
        while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
            transfer is complete
        IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
            LOWgyro_temp);
        while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
            transfer is complete
        response = IORD(GYRO_X_BASE,0);
        uint8_t LOWgyro_x = (response & 0xff); //get 8 LSB
        uint8_t HIGHgyro_x = ((response>>8) & 0xff);
            //right shift 8 places and get 8 LSB
        IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
            HIGHgyro_x);
        while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
            transfer is complete
        IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
            LOWgyro_x);
        while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
            transfer is complete
        response = IORD(MAG_Y_BASE,0);
        uint8_t LOWgyro_y = (response & 0xff); //get 8 LSB
        uint8_t HIGHgyro_y = ((response>>8) & 0xff);
            //right shift 8 places and get 8 LSB
        IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
            HIGHgyro_y);
        while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
            transfer is complete
        IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
            LOWgyro_y);
        while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
            transfer is complete
        response = IORD(MAG_Z_BASE,0);
        uint8_t LOWgyro_z = (response & 0xff); //get 8 LSB
        uint8_t HIGHgyro_z = ((response>>8) & 0xff);
            //right shift 8 places and get 8 LSB
```

```
            IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                HIGHgyro_z);
            while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                transfer is complete
            IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                LOWgyro_z);
            while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                transfer is complete

        //send current data
            response = IORD(INA_X_BASE,0);
            uint8_t LOWina_x = (response & 0xff); //get 8 LSB
            uint8_t HIGHina_x = ((response>>8) & 0xff);
                //right shift 8 places and get 8 LSB
            IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                HIGHina_x);
            while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                transfer is complete
            IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                LOWina_x);
            while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                transfer is complete
            response = IORD(INA_Y_BASE,0);
            uint8_t LOWina_y = (response & 0xff); //get 8 LSB
            uint8_t HIGHina_y = ((response>>8) & 0xff);
                //right shift 8 places and get 8 LSB
            IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                HIGHina_y);
            while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                transfer is complete
            IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                LOWina_y);
            while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                transfer is complete
            response = IORD(INA_Z_BASE,0);
            uint8_t LOWina_z = (response & 0xff); //get 8 LSB
            uint8_t HIGHina_z = ((response>>8) & 0xff);
                //right shift 8 places and get 8 LSB
            IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                HIGHina_z);
            while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                transfer is complete
```

```
                IOWR_ALTERA_AVALON_PIO_DATA(OBC_DATA_BASE,
                    LOWina_z);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete

        break;

//restart
        case 0x55:
                //Send command in return
                IOWR(OBC_DATA_BASE, 0, 0x55);
                while(IORD(BUSY_SLAVE_BASE,0)); //Wait until
                    transfer is complete
                //Reset the system
                IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,1);
                IORD(RESET_ITG_BASE,0);
                IOWR_ALTERA_AVALON_PIO_DATA(RESET_ITG_BASE,0);
        break;

//activate magnetic coils
        case 0x87:
        coil = IORD(COIL_CHOICE_BASE,0);

        if (coil == 0x1) {//X coil
                initina();
                IOWR_ALTERA_AVALON_PIO_DATA(START_INA_BASE, 0);
                //printf("coil 1\n");
                IOWR_ALTERA_AVALON_PIO_DATA(COILS_ENABLE_BASE,
                    0x0F);
                IOWR(OBC_DATA_BASE, 0, 0x87);
                current();
        }
        else if (coil == 0x2){ //Y coil
                printf("coil 2\n");
                IOWR_ALTERA_AVALON_PIO_DATA(COILS_ENABLE_BASE,
                    0x1F);
                IOWR(OBC_DATA_BASE, 0, 0x4B);
                current();
        }
        else if (coil == 0x3){ //Z coil
                printf("coil 3\n");
                IOWR_ALTERA_AVALON_PIO_DATA(COILS_ENABLE_BASE,
                    0x2F);
```

```
                        IOWR(OBC_DATA_BASE, 0, 0x2B);
                        current();
                }
                else if (coil == 0x0){ //Turn of coils //break
                        IOWR_ALTERA_AVALON_PIO_DATA(COILS_ENABLE_BASE,
                            0x0);
                        IOWR(OBC_DATA_BASE, 0, 0x00);
                        current();
                }
                break;

        // ADCS enable
                case 0x99:
                        //return the corresponding command:
                        IOWR(OBC_DATA_BASE,0, 0x99);
                break;

        // ADCS disable
                case 0xAA:
                //return the corresponding command:
                        IOWR(OBC_DATA_BASE,0, 0xAA);
                break;

        // ADCS get attitude
                case 0xB4:
                //return the corresponding command:
                        IOWR(OBC_DATA_BASE,0, 0xB4;
                break;

        }//switch end
} //if end
else {
        gyro();
        magnetometer();
        current();

}

 } //while end
}  //Main end!
```

Appendix D

Matlab Code

D.1 Kalman Filter

```
%%%%KalmanFilterCB_tb.m Author: Cecilie Bjelbøle
%%%%Test of the filter KalmanFilterCB.m
%%%%The file creates 3 sin signal with added noise
%% Clear Workspace
clear all
close all
clc

%% Create a input signal;
t = [-3*pi : 0.1 : 3*pi];

x = 2 + sin(t);
y = 8 + cos(t)+sin(t);
z = -1+ sin(t);

%% Add noise:

x1 = 0.2.*randn(length(x) ,1);
X = x1'+ x;
y1 = 0.2.*randn(length(x) ,1);
Y = y1'+ y;
z1 = 0.3.*randn(length(x) ,1);
Z = z1'+ z;
Measurement = [X' Y' Z'];

error = 0.12;
```

```
%% Performing the Kalman filtering
for j = 1:length(Measurement)
     [OHM_X(j),OHM_Y(j),OHM_Z(j)] = KalmanFilterCB(X(j), Y(j), Z(j));
 end
%% Plot
plot(OHM_X','-k')
hold on
plot(OHM_Y','-k')
plot(OHM_Z','-k')
%plot(OHM')
hold on
plot(Measurement)
xlabel('Time (samples)','Interpreter','latex','fontsize',12);
ylabel('Amplitude','Interpreter','latex','fontsize',12);
```

```
function [ Ohm_est_out_x, Ohm_est_out_y, Ohm_est_out_z] =
    KalmanFilterCB(X, Y, Z )
%Kalman filter for angular rate estimation

ohm_unf = [X; Y; Z;];
R_ohm = [0.73^2 0 0; 0 1.023^2 0 ; 0 0 1.44^2];
Q_ohm = eye(3)*0.82;

%initial conditions
persistent P_ohm_prev Ohm_est;
if isempty(Ohm_est)
    %Ohm_est = zeros(3,1); %x_est
    Ohm_est = ohm_unf;%
    P_ohm_prev = zeros(3,3); %p_est
end

    ohm_apriori = Ohm_est;  %x_prd = x_est
    P_apriori = P_ohm_prev + Q_ohm; %p_prd = p_est + Q
                                    % z_prd = x_prd
%Kalman gain: mlhdlc_backslash is a replacement function for '\'
%K = P_apriori*inv(P_apriori + R_ohm);
    S = P_apriori + R_ohm;
    B = P_apriori;
    K = mlhdlc_backslash(S, B)';

%Ohm_est is our state:
   % x_est = x_prd + klm_gain * (z - z_prd);
   Ohm_est = ohm_apriori + K*(ohm_unf - ohm_apriori );
```

```
% Update for next cycle
    P_ohm_prev = (eye(3) - K) * P_apriori;
    %Ohm_est_prev = Ohm_est;

    Ohm_est_out_x = Ohm_est(1);
    Ohm_est_out_y = Ohm_est(2);
    Ohm_est_out_z = Ohm_est(3);


end
```

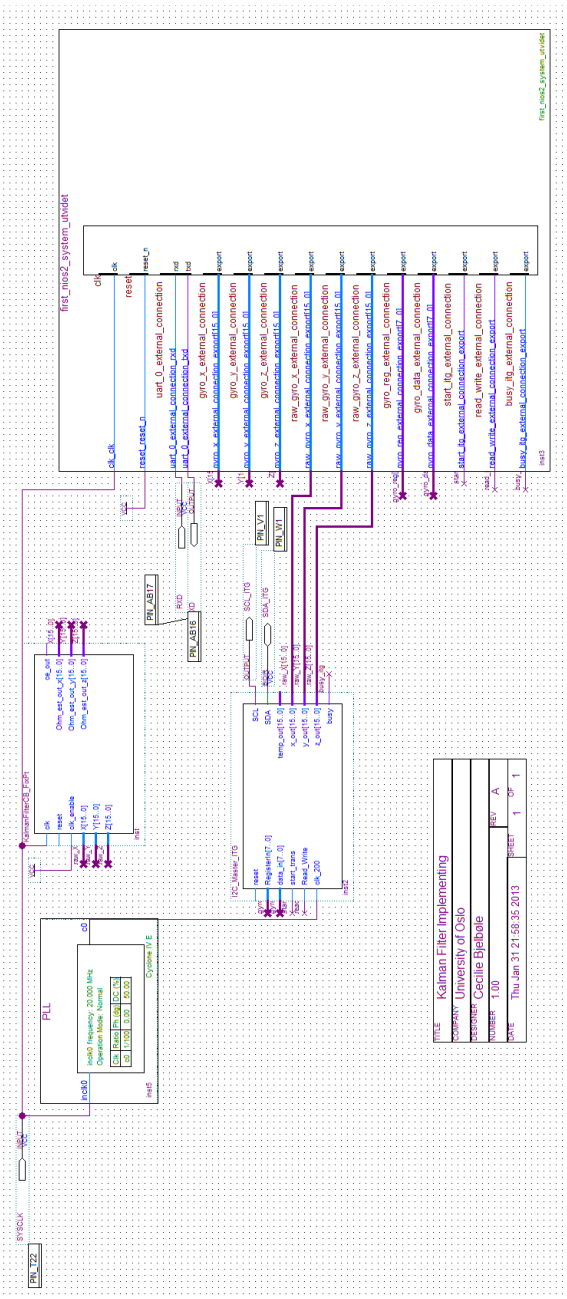
```
function x = mlhdlc_backslash(A,b)
%Divider

% Parse input
[rA,~] = size(A);
[~, cb] = size(b);
N = rA;
D = cb;

% Initialize x
x = zeros(N,D);

% Compute LU decomposition of A
[L, U] = lu_decomp(A);

% Solve equation x = A\b for every column of b through forward and
% backward substitution
for col = 1:D
    bcol = b(:,col);

    % Forward substitution
    y = fsubs(L, bcol);

    % Backward substitution
    x(:,col) = bsubs(U, y);

end

end

function [L,U]=lu_decomp(A)
```

```
N=size(A,1);
U=A; L=eye(N);

for n=1:N-1
    piv = U(n,n);
    for k=n+1:N
        mult = U(k,n)/piv;
        U(k,:) = -mult*U(n,:) + U(k,:);
        L(k,n) = mult;
    end
end
end


function y = fsubs(L,b)

N = numel(b);
y = zeros(N,1);

% Forward substitution
y(1) = b(1)/L(1,1);
for n = 2:N
    acc = 0;
    for k = 1:n-1
        acc = acc + y(k)*L(n,k);
    end
    y(n) = (b(n) - acc)/L(n,n);
end

end

function x = bsubs(U,y)

N = numel(y);
x = zeros(N,1);

% Backward substitution
d1 = U(N,N);
n1 = y(N);
x(N) = n1/d1;

for n = N-1:-1:1
```

```
    acc = 0;
    for k = n:N
        acc = acc + x(k)*U(n,k);
    end
    n2 = (y(n) - acc);
    d2 = U(n,n);
    x(n) = n2/d2;
end

end
```

Appendix E

Kalman Filter Implementation


first_nios2_system_inst
clk
reset_n
uart_0_external_connection
gyro_x_external_connection
gyro_y_external_connection
gyro_z_external_connection
raw_gyro_x_external_connection
raw_gyro_y_external_connection
raw_gyro_z_external_connection
gyro_reg_external_connection
gyro_data_external_connection
start_ltg_external_connection
read_write_external_connection
busy_ltg_external_connection
PLL
inclk0 frequency: 20.000 MHz
Operation Mode: Normal
c0 1:100 2.00 50.00
Cyclone IV E
I2C_Master_ITG
KalmanFilterOE_Prof1
PIN_T22
PIN_AB17
PIN_AB16
PIN_V11
PIN_V1
Kalman Filter Implementing
University of Oslo
Cecilie Bjelbole
1.00
A
Thu Jan 31 21:58:35 2013
1
1